Ohjelmavirheiden automaattisesta korjaamisesta

Marko Elo

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Marraskuu 2016

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Marko Elo
Pro gradu -tutkielma, 69 sivua
Marraskuu 2016

Käyttökohteita automaatiolle löytyy yhä enemmän laskentatehokkuuden ja algoritmien tehokkuuden parantuessa. Algoritmit ovat nyt riittävän tehokkaita käsittelemään suurtakin hakuavaruutta, jolloin niitä voidaan käyttää automaattiseen ohjelmavirheiden korjaukseen.

Tässä tutkielmassa esitellään kaksi menetelmää automaattiseen ohjelmavirheiden korjaukseen ja pohditaan automaattisesti tapahtuvan ohjelmavirheiden korjauksen mielekkyyttä. Käsiteltävänä on geneettiseen algoritmiin perustuva toteutus GenProg ja semanttiseen analyysiin perustuvan menetelmän toteutukset SemFix ja DirectFix.

Tutkielmasta selviää, että käsintehtävää virheiden korjausta ei voi täysin korvata. Automaattinen virheiden korjaus voisi kuitenkin olla osa projekteissa tapahtuvaa automaattista testausta.

Avainsanat ja -sanonnat: automaattinen ohjelman korjaus, geneettinen algoritmi, ohjelman syntetisointi, formaalit menetelmät.

Sisällys

1. Johdanto

Tässä pro gradu -tutkielmassa analysoin automaattiseen ohjelmavirheiden korjaamiseen kehitettyjä menetelmiä kirjallisuutta tutkimalla. Tutkittavat menetelmät olen rajannut vain staattisiin menetelmiin, eikä ohjelmien ajonaikaiseen korjaamiseen kehitettyjä menetelmiä käsitellä.

Tietokoneohjelmat ovat enemmän taikka vähemmän virheellisiä. Ne ovat ihmisten tekemiä, ja täten niihin voi tulla inhimillinen virhe. Tyyppiesimerkki on C-kielen if-lause, kun ehto-osassa `if(x=1)` ei tapahdukaan suoraan vertausta arvoon 1, vaan muuttujaan `x` sijoitetaan arvo 1. Toinen syy ohjelmien virheille on, että ne ohjelmoidaan toimimaan oikein, jos niille annettu syöte on sitä, mitä ohjelman syötteen pitäisi olla. Ohjelmista etsitään tietoturva-aukkoja antamalla niille yli- tai alimittaista dataa, tai merkkejä, joita normaalisti ei kuuluisi antaa. Ohjelmoijat eivät aina ole varautuneet tahalliseen ohjelman väärinkäyttöön, koska jo ohjelman saaminen toimivaksi tarkoituksenmukaisilla syötteillä on työlästä. Koska näkökulma ohjelman kehittämiselle on sen toiminta halutussa tilassa, eivät ohjelmoijat ota aina huomioon yllättäen järjestelmässä esiintyvää virheellistä tilannetta, kuten esimerkiksi odottamatonta muistin vähentymistä. Tälläinen tilanne tulee esimerkiksi silloin, kun korkeammalla prioriteetilla oleva ohjelma varaa järjestelmän resursseja taustalla, ja virheen sisältävä ohjelma ei osaakaan varautua siihen, että sen jo varaama resurssi ei enää kuulukaan sen käyttöön.

Ohjelmien luomista pyrittiin automatisoimaan jo 1980-luvulla ylläpitoon liittyvien ongelmien vuoksi. Automatisoinnissa pyrittiin luomaan ohjelma spesifikaation pohjalta. Tällöin saatiin aikaiseksi työkalu, jolla spesifikaatiossa olevia virheitä pystyttiin löytämään automaattisesti. Myös spesifikaation virheitä saatiin automaattisesti korjauttua Replay-sovelluksella. Balzerin [1985] katsauksessa ei kuitenkaan mainittu mahdollisuutta itse lähdekooditason automaattiseen korjaukseen.

Automaattinen ohjelmointi on kiehtonut tietojenkäsittelytieteen tutkijoita ainakin viimeiset 50 vuotta, mutta virheiden korjaus ei tarjoa automaattista ohjelmointia luomalla uusia ohjelmia tyhjästä. Automaattinen virheiden korjaus kuitenkin auttaa vanhojen ohjelmien pienenä kehittymisenä [Weimer et al., 2010].

Vuonna 2005 on kuvailtu menetelmä, joka käyttää LTL-spesifikaatioita (*linear temporal logic*) äärellisten tilojen ohjelmien (*finite state programs*) korjaamiseen. Menetelmässä korjauksen hakeminen on pelkistetty peliksi ja voittava strategia vastaa onnistunutta korjausta [Nguyen et al., 2013].

Vuonna 2008 on myös kuvailtu menetelmä, joka automaattisesti loi ohjelmavirheitä hyödyntäviä työkaluja ohjelmapaikkauksista. Kyseisen kaltaiset tutkimukset aiheuttavat tietenkin huolta siitä, että ne saattavat päätyä menetelmiä hyödyntävien hyökkääjien käyttöön [Goues et al., 2012].

Aikaisemmissa töissä GenProgin kehittäjät kehittivät algoritmin, joka automaattisesti korjasi ohjelmia, joille spesifikaatio oli saatavilla [Weimer et al., 2010].

Ensimmäinen toimiva menetelmä, jota on kokeiltu merkittävään määrään käytössä olevia ohjelmia, joihin spesifikaatiota ei ole saatavilla ja niissä oleviin todellisiin virheisiin, on geneettinen algoritmi [Goues et al., 2012].

Ohjelmoijat testaavat jonkin verran manuaalisesti koodinsa toimintaa ennen kuin koodi siirtyy regressiotestaukseen. Testaus on suhteellisen helppo automatisoida ja toimii manuaalista testausta nopeammin. Automaattisessa testauksessa on ohjelmoija kirjoittanut testausskriptin, joka lähettää testattavalle ohjelmalle halutut syötteet ja usein myös satunnaisesti tuotettua syötettä. Syötteestä ja ohjelman antamasta palautteesta tulee testiraportti. Halutuille syötteille tiedetään yleensä haluttu tulos ja testi epäonnistuu, jos ohjelman antama palaute syötteelle ei vastaa haluttua tulosta. Satunnaisella syötteellä voidaan varmistua, että ohjelma toimii edelleen normaalisti. Ohjelma voi lakata toimimasta, jolloin testi kyseisellä syötteellä on merkitty epäonnistuneeksi. Ohjelma voi myös alkaa toimia virheellisesti satunnaisen syötteen jälkeen, jolloin ohjelma palauttaakin virheellisiä vastauksia halutuilla testisyötteillä. Kun testi on epäonnistunut, ohjelma palautuu ohjelmoijalle korjattavaksi.

Virheenpaikannus ja korjaus on helppoa, jos viimeksi tehty ohjelman muutos on aiheuttanut aina toistettavissa olevan virheen. Virhe saadaan suoraan paikannettua muuttuneeseen koodiin, tai ainakin muuttunut koodi saa esille koodissa aiemmin olleen virheen. Hankalampia tapauksia ovat harvemmin toistuvat virheet ja sellaiset, jotka ovat tulleet esille ohjelman loppukäyttäjien virheilmoituksista. Tällöin ohjelmoija yrittää toistaa virheen debug-optioilla käännetyllä ohjelmalla ja tarkastelee debug-version tuottamaa jäljityslogia ohjelman kulusta, tilasta ja muuttujien arvoista. Tämä toimenpide on aikaa vievin osa virheen korjauksesta, ja sillä saadaan vasta paikannettua virhe ja ymmärrettyä, mistä virhe johtuu. Joissakin tapauksissa joudutaan ohjelmaa ajamaan debuggerilla ja asettamalla ohjelman suoritukselle pysähdyspisteitä eli breakpointeja. Pysähdyspisteistä ohjelmaa ajetaan yleensä debuggerilla askel kerrallaan. Askelluksessa voidaan seurata muuttujien ja prosesssorin rekisterien arvoja ja näitä arvoja verrataan oikein toimivan ja virheellisesti toimivan tapauksen välillä. Tämänlaiseen työhön voi kulua työpäivä vain yhden virheen korjaamiseen. Paikallistetun ja ymmärretyn virheen korjaus, ohjelmoijan suorittama alustava testaus ja raportti korjatusta virheestä eivät yleensä kestä yhtä tuntia kauempaa. Virheenkorjaus voi kuitenkin sisältää inhimillisen virheen, kuten alkuperäinenkin ohjelma, ja siten aiheuttaa regressiota. Tämän prosessin automatisointi vähentäisi inhimillisiä virheitä ja siten mahdollista regressiota.

Jäljitykseen ja virheen paikantamiseen keskittyneet projektit koittavat selventää virheitä ja helpottaa niiden tutkimista (*debugging*). Nämä menetelmät ovat myös apuna automaattisessa virheiden korjauksessa [Weimer et al., 2010].

Automaattinen virheiden korjaus on automaattisesti ajettujen epäonnistuneita testitapauksia aiheuttaneiden virheiden korjaamista ohjelmallisesti, ilman manuaalista työtä. Virheiden korjaukseen siirrytään, kun testitapaus epäonnistuu. Virheen korjaus on kaksivaiheinen. Aluksi haarukoidaan todennäköisin virheen paikka jollakin menetelmällä. Jälkimmäisessä vaiheessa yritetään löytää sopiva korjaus kyseiseen kohtaan. Korjauksen toimivuus testataan. Toimivan korjauksen jälkeen

saatetaan joillakin menetelmillä vielä yrittää korjauksen yksinkertaistamista siten, että korjaus aiheuttaa mahdollisimman vähän muutoksia alkuperäiseen ohjelmaan ja on ihmisen luettavissa.

Tuloksena ollaan saavutettu noin 50 prosentin virheiden korjaus yleisesti käytössä oleviin tuotteisiin. Tulokset ovat erittäin kustannustehokkaita verrattuna manuaaliseen virheiden korjaamiseen. Automaattisessa virheiden korjauksessa on kuitenkin rajoitteita, jotka estävät tietyntyyppisten virheiden korjaamista, jolloin se ei voi korvata kokonaan manuaalista virheiden korjausta.

Luvussa 2 on katsaus termeihin sekä määrittelyihin, jotka helpottavat tämän tutkielman lukemista. Luvussa 3 käydään lävitse komponenttipohjaista ohjelman syntetisointia ja siihen liittyvää teoriaa. Automaattiseen virheidenkorjaukseen liittyviä ohjelmien testausmenetelmiä esitellään luvussa 4. Luvussa 5 käydään läpi automaattisia korjausmenetelmiä. Luvussa 6 käydään läpi virheen paikannusmenetelmiä. Luvussa 7 analysoidaan virheenkorjausmenetelmillä saatuja ja ilmoitettuja tuloksia ja tuloksiin vaikuttavia seikkoja sekä pohditaan jatkotutkimuksien mielekkyyttä.

2. Termit ja käsitteet automaattisessa virheiden korjauksessa

Tässä luvussa esittelen esille tulevia termejä ja käsitteitä sekä niiden määritelmiä ja tarvittaessa annan alkuperäisen englanninkielisen termin. Näitä käsitteitä käytetään automaatisesta virheenkorjauksesta kertovissa artikkeleissa, ja ne kuvaavat toteutuksissa käytettyjä tietorakenteita ja operaatioiden suorittamista tietorakenteissa. Käsitteet helpottavat tämän työn lukemista luvusta 3 eteenpäin, mikäli ne eivät ole lukijalle entuudestaan tuttuja.

2.1. Puurakenteinen erottelu

Puurakenteinen erottelu (tree-structured differencing) on alun perin kehitetty XML-dokumenttien muutosten seuraamiseen. Kehityksen tuloksena oli menetelmä nimeltä diffX, joka vertasi kahta XML-dokumenttia keskenään ja tuotti tulokseksi muutosoperaatioita sisältävän skriptin, jolla ensimmäisestä dokumentista sai aikaiseksi jälkimmäisen. Menetelmä lähtee XML-dokumentin lukemisesta puurakenteeksi siten, että tagit ovat solmuja. Sisältöä voi olla myös solmuissa, eikä vain lehdissä. Menetelmä tunnistaa samanlaiset rakennepuun osaset. Tavoitteena on tunnistaa suurimmat toisiaan vastaavat osaset rakenteeltaan ja sisällöltään. Kahden dokumentin välisiä eroja lähdetään etsimään vanhimman puun juuresta, ja prosessi jatkuu, kunnes puun kaikki solmut on käyty lävitse. Toisen puun kaikki tarkasteltavana olevaan solmuun sopivat solmut tarkastetaan rekursiivisesti ylhäältä alas, kunnes yhteensopivia lapsisolmuja ei enää ole, tai kohdataan jo tarkastettu solmu. Löydetyt yhteensopivat puut lisätään kuvausten (mapping) joukkoon [Al-Ekram et al., 2005].

Saaduista kuvauksista kehitetään skripti. Skripti poistaa alkuperäisestä versiosta solmut, joita ei löydy uudemmasta versiosta, sekä lisää vain uudemmasta löytyvät solmut alkuperäiseen versioon. Molemmista versiosta löytyvät solmut, joiden vanhemmat eivät ole samat tai niiden sijainti dokumentissa ei ole sama, siirretään samaan kohtaan kuin uudemmassa versiossa. Algoritmin kuvaamiseksi käytetään dokumentin uudemmasta versiosta merkintää T_2 ja vanhemmasta merkintää T_1. Algoritmi käy läpi T_2:n solmut ylhäältä alaspäin. Niille solmuille, joille ei löydy vastaavuutta T_1:sta, skriptiin lisätään lisäysoperaatio. Niille solmuille, joille löytyy vastaavuus T_1:ssä, mutta joilla on eri vanhempi tai eri sijainti verrattuna solmun sijaintiin T_2:ssa, skriptiin lisätään siirto-operaatio. Tasojärjestelty puun läpikäyminen varmistaa, että solmun vanhempi on oikeassa paikassa ennen kuin siihen siirretään tai lisätään lapsi. Kun kaikki solmut, joihin löytyy vastaavuus, on läpikäyty, T_1:ssä ilman vastaavuuksia olevat solmut sijaitsevat puun pohjalla. Näitä varten skriptiin lisätään poisto-operaatio. Poisto-operaatioiden lisäys tapahtuu käymällä ylhäältä alas tasojärjestyksessä T_1:stä kuvaavaa puuta, jonka juurisolmu on ensimmäinen solmu, jolle ei löydy vastaavuutta. DiffX:n kehittäjät mainitsevat algoritminsa aikavaativuudeksi n^2, kun n on dokumentissa olevien solmujen määrä [Al-Ekram et al., 2005].

2.2. Ilmaisut, lausumat ja formaali spesifikaatio

Lausuma (*statement*) on pienin mahdollinen imperatiivisten ohjelmointikielten elementti, joka ilmaisee jonkin toimenpiteen suorittamista. Sillä voi olla sisäisiä komponentteja kuten ilmaisuja

(*expressions*). Useimmissa kielissä lausuma ei palauta arvoa, mutta sillä on sivuvaikutuksia. Esimerkiksi sijoitus lausuma `A=A+5;` lisää muuttujan `A` arvoon 5, mutta se ei siis palauta mitään arvoa. Ilmaisu palauttaa aina arvon, mutta sillä ei ole sivuvaikutuksia, kuten esimerkiksi vertailu `A<5` palauttaa tuloksen `tosi/epätosi` mutta ei muuta muuttujan `A` sisältöä.

Formaali spesifikaatio (*formal specification*) on ohjelmakoodin matemaattisesti tarkasti esitetty määrittely. Formaali on suomeksi muodollinen (*formal*), mutta matematiikassa erityismerkitys ”mekaanisesti (tietokoneella) tarkistettavissa oleva”. Matematiikassa on myös määrittely ”erityisen täsmällisesti tehty”. Formaali spesifikaatio on eräs formaalien menetelmien väline [Kaijanaho, 2006].

Esimerkiksi funktio, joka muuttaa annetun luvun yhtenäisen ykkösiä sisältävän oikeanpuolimmaisen jonon bitit nolliksi voidaan ilmaista predikaattilogiikalla, jossa muuttuja n merkitsee annetun bittijonon pituutta, seuraavasti:

$$\exists i, j.\ \{\ 0 \le i, j < n \wedge (\forall k.\ j \le k \le i \Rightarrow x[k] = 1)$$
$$\wedge\ (\forall k.\ 0 \le k < j \Rightarrow x[k] = 0)$$
$$\wedge\ (x[i+1] = 0 \vee i = n - 1)$$
$$\wedge\ (\forall k.\ i < k < n \Rightarrow x[k] = y[k])$$
$$\wedge\ (\forall k.\ 0 \le k \le i \Rightarrow y[k] = 0)\ \}.$$

Hyvin tiiviisti C-ohjelmointikielellä kirjoitettuna voidaan sama funktio ilmaista seuraavasti:

```
1 turnOffRightMostOneBitString (x)
2 { t1 = x-1; t2 = (x | t1); t3 = t2+1;
3 t4 = (t3 & x); return t4; }
```

[Jha et al., 2010].

Propositiologiikassa eli lauselogiikassa käytettävät symbolit on esitelty taulukossa 1. Propositiologiikassa literaali (*literal, fact*) on totuusfunktion muuttuja, esimerkiksi x tai sen negaatio ¬x. Peruskonjunktio on literaali tai literaalien konjunktio, esimerkiksi $x \wedge \neg y$. Perusdisjunktio on myös literaali, esimerkiksi x, kuten peruskonjunktiossakin, mutta literaalien disjunktio on $x \vee \neg y$. On olemassa disjunktiivinen normaalimuoto DNF (*disjunctive normal form*) ja konjunktiivinen normaali muoto CNF (*conjunctive normal form*). CNF muotoisia klausuuleja käytetään toteutuvuusongelmien (SAT, katso kohta 2.4) ratkaisemisessa.

Operaatio	Standardi	Vaihtoehtoiset symbolit	C/C++/Java…
1, tosi	$\top$	1	true
0, epätosi	$\bot$	0	false
Negaatio, ei	$\neg x$	$\bar{x}$, ~x, −x	!x
Konjunktio, ja	$x \wedge y$	x ·y, x & y	x && y
Disjunktio, tai	$x \vee y$	x + y, $x \supset y$, $x \Rightarrow y$	x || y
Ekvivalenssi, yhtäpitävyys	$x \leftrightarrow y$	x = y, $x \equiv y$, $x \Leftrightarrow y$	x == y
Implikaatio, seuraus	$x \Rightarrow y$	$\supset$	
Poissulkeva disjunktio	$x \oplus y$	$\veebar$	a ^ b (a XOR b) bittioperaatio, a != b, looginen
Tautologia	$\top$		

Taulukko 1. Propositiologiikan symbolit.

Usein propositiologiikka ei ole riittävä ja käytetään predikaattilogiikkaa. Propositiot ovat suljettuja lauseita eli ne ovat aina joko tosia tai epätosia. Tarvitaan myös avoimia lauseita eli predikaatteja, joiden totuusarvo riippuu tarkasteltavana olevista muuttujista. Kvanttoreiden avulla on mahdollista saada avoimia lauseita suljettua ja määritettyä niille totuusarvot [Kaarakka ja Lätti, 2006].

Formaalissa spesifikaatiossa saatetaan tarvita myös joukko-opillisia operaatioita ja merkintöjä, kuten esimerkiksi yhdiste $\cup$, leikkaus $\cap$ ja $\setminus$ komplementointi. Merkintä $\subseteq$ tarkoittaa osajoukkoa. Validi, kelvollinen tai pitävä seuraus (*valid consequence*) symbolia $\vDash$ käytetään mallien pitävyyden merkitsemiseen.

Kun on olemassa kaavan B deduktio premissijoukosta S, niin sanomme kaavan B olevan dedusoitavissa eli johdettavissa joukosta S ja merkitsemme $S \vdash B$.

Formaalilla spesifikaatiolla tehtyjä lauseita voidaan käyttää deklaratiivisessa ohjelmoinnissa (*declarative programming*). Logiikkaohjelmointi, joka on deklaratiivista ohjelmointia, voi tuottaa suoritettavaa tai suorittavaa ohjelmakoodia loogisilla lauseilla tehdystä spesifikaatiosta, kun se on muunnettu käytetyn kääntäjän hyväksymäksi syntaksiksi. Lähteestä riippuen myös funktionaalinen ohjelmointi kuuluu deklaratiiviseen ohjelmointiin logiikkaohjelmoinnin lisäksi. Funktionaalista ohjelmointia käytettäessä ei riitä pelkkä syntaksin muunnos. Edellämainituista kahdesta ohjelmointitavasta hieman etäältä tuntuvia deklaratiivista ohjelmointikieliä ovat esimerkiksi SQL, HTML, XAML ja QML. Edellä mainituista HTML, XAML ja QML kuuluvat deklaratiivisiin merkintäkieliin (*declarative markup language*).

2.3. Sopimuspohjainen ohjelmointi

Sopimuspohjainen ohjelmointi on alun perin Eiffel-kielen kehittäjien luoma ohjelmoinnin suunnittelutapa, Design By Contract. Pääasiallinen idea on, että luokalla ja sen asiakkailla (*client*) on

”sopimus” (*contract*) keskenään. Asiakkaan täytyy taata tietyt ehdot (*condition*) ennen käyttämänsä luokan metodien kutsumista. Sopimuksessa määritetään metodin kutsujan eli asiakkaan täyttämät esiehdot (*preconditions*) ja jälkiehdot (*postconditions*), jotka luokan on taattava metodin suorituksen loputtua. Näiden esi- ja jälkiehtojen käyttämisestä ohjelman spesifikaatiossa on mainittu Hoaren artikkelissa vuodelta 1969 formaalista tarkistamisesta (*formal verification*) [Leavens and Cheon, 2006].

Esimerkiksi Java modeling language (JML) lukee sopimusten määrittelyt lähdekoodissa olevista kommenttikentistä, jotka on annettu erityisillä huomautus kommenteilla (*annotation comments*). Taulukon 2 esimerkeistä huomaa, miten JML on laajentanut Java-kielen ominaisuuksia esimerkiksi predikaattilogiikasta tutuilla kvanttoreilla. Näillä Java-kieltä laajentavilla spesifikaatio rakenteilla (*specification constructs*) on JML:ssä pyritty laajentamaan ilmaisukykyä käyttäytymiseen liittyvien spesifikaatioiden kirjoittamista varten [Leavens and Cheon, 2006].

`//@ requires x >= 0.0;`
`/*@ ensures JMLDouble`
`@ .approximatelyEqualTo`
`@ (x, \result * \result, eps);`
`@*/`
`public static double sqrt(double x) {`
`/*...*/`
`}`
	Esiehdot (requires), jotka annetaan `requires`-sanalla ovat tässä yksinkertaiset. Jälkiehdoissa (postconditions), avainsana `ensures`, sisältävät kutsun staattiseen funktioon `.approximatelyEqualTo` oikeellisuuden tarkistamista varten.

`/*@ requires a != null`
`@ && (\forall int i;`
`@ 0 < i && i < a.length;`
`@ a[i-1] <= a[i]);`
`@*/`
`int binarySearch(int[] a, int x) {`
`// ...`
`}`
	Esiehdoissa tarvitaan kvanttoria. Kvanttori on tekstinä `\forall`. Kommenttilohkon lopussa oleva `<=` ei ole implikaatio vaan sillä on sama merkitys kuin itse käännettävässä java-koodissa.

Taulukko 2. Esimerkkejä JML:n käytöstä kommenttilohkossa.

JML käyttää Javan syntaksia väittämissä (*assertion*) käytettäviin predikaatteihin (*predicate*), kuten esi- ja jälkiehtoihin ja pysyväisväittämiin (*invariant*). Javan merkintöjen käytön etuna on, että sen oppiminen on ohjelmoijille helpompaa kuin erikoistarkoitettujen matemaattisia merkintöjä sisältävien kielten oppiminen. Näitä sopimuksia voidaan soveltaa formaalissa tarkistamisessa käyttämällä teoreematodistajia (*theorem provers*). Spesifikaatioiden ilmauksilla on JML:ssä myös rajoituksia Javaan verrattuna [Leavens and Cheon, 2006].

2.4. Toteutuvuus modulo teoriat

Toteutuvuus modulo teorioita (SMT, *satisfiability modulo theories*) ja niiden suppeampaa muotoa lauselogiikan toteutuvuusongelmaa SAT (*propositional satisfiability problem*, *boolean satisfiability*, *boolean satisfiability problem*) käytetään formaalisti syötettyjen kaavojen toteutuvuuden selvittämiseen, eli haetaan vastausta toteutuvuusongelman ratkeavuuteen. Ongelma on käytännössä ratkeava (*tractable*), jos sille on olemassa polynomisessa ajassa toimiva ratkaisualgoritmi. Työläille ongelmille (*intractable*) on olemassa vain eksponentiaalisen ajan vaativia ratkaisualgoritmeja.

Toteutuvuusongelma on aikavaativuudeltaan NP-täydellinen (*nondeterministic polynomial time*), eli ongelma ratkeaa polynomisessa ajassa epädeterministisellä Turingin koneella. Aikavaativuus kasvaa eksponentiaaliseksi deterministisellä Turingin koneella. Todellisuudessa kaikki algoritmien ratkojat eli tietokoneet ovat deterministisiä Turingin koneita. Turingin koneista, Turingin koneiden tyypeistä, aikavaativuusluokista ja niiden määrittelyistä sekä todistuksesta, että toteutuvuusongelma on NP-täydellinen voi lukea tarkemmin esimerkiksi Gurarin kirjasta [1989].

Toteutuvuusongelmassa kysytään, onko annettu Boolean kaava (*boolean expression*) toteutuva. Esimerkiksi Boolean kaava E = x2 ∧ x3 ∨ (¬x1 ∧ x2) on toteutuva aina, kun sijoitetaan x2=1 ja x3=1 tai jos sijoitetaan x1=0 ja x2=1. Esimerkki ei-toteutuvasta Boolean kaavasta x∧¬x [Gurari, 1989].

Esimerkki SMT-LIB-dokumentaatiosta havainnollistaa muuttujien lukualueista johtuvia rajoitteita toteutuvuuteen. Yhtälöpari x+2*y=20 ja x-y=2 on toteutuva kokonaisluvuilla (*integer*) ja sijoituksilla x=8 ja y=6. Jos jälkimmäinen yhtälö muutetaan muotoon x-y=3, niin yhtälöpari ei ole enää toteutuva kokonaisluvuilla. Yhtälöpari on toteutuva reaaliluvuilla y=17/3 ja x=26/3. SMT-LIB voi pelkän toteutuvuuden lisäksi antaa muuttujien arvot, joilla haluttu kaava toteutuu [Cok, 2013].

3SAT on SAT ongelman rajoittuneempi muoto. Lausekalkyylin kaava (boolen kaava), tai klausuuli (*clause*) on CNF-muodossa, jos se on disjunktio-klausuulien konjunktio, esimerkiksi kaava $E=C_1 \wedge C_2 \wedge \ldots \wedge C_m$, missä kukin tekijä C_i on disjunktio $C_i=a_{i1} \vee a_{i2} \vee \ldots \vee a_{ir}$, jossa termit a_{ij} ovat literaaleja. Kaava on k-CNF, jos sen jokaisessa klausuulissa on tasan k-literaalia. Boolean kaava on klausuuli, jos se on literaalien disjunktio. Kieli 2SAT kuuluu aikavaativuusluokkaan P, eli sen ratkaisemiseksi on olemassa polynomisessa ajassa toimiva deterministinen algoritmi [Gurari, 1989].

Tarkastellaan klausuulia $l_1 \vee \ldots \vee l_k$, missä jokainen *li* on literaali. Klausuulissa ∨ on disjunktio eli looginen OR-operaatio. Lyhyemmin klausuuli voidaan kirjoittaa $\bigvee_{i=1}^{k} l_i$, jossa *k* on klausuulin pituus. Klausuulin muuttujaan *k* liittyy seuraavat määrittelyt: *k*=1 yksikköklausuuli (*unit clause*) ja *k*=2 binääriklausuuli (*binary clause*). Jatketaan esimerkillä CNF-kaavasta $C_1 \wedge \ldots \wedge C_m$, jossa jokainen C_i on klausuuli. Loogista operaatiota ∧ eli AND-operaatiota kutsutaan konjunktioksi. Lyhyemmin CNF-kaava voidaan kirjoittaa seuraavasti $\bigwedge_{i=1}^{m} C_i$. Näitä kaavoja tarkastellaan yleensä klausuulien joukkoina [Järvisalo, 2016].

SAT-ratkaisijat mahdollistavat tehokkaat NP-ongelmien ratkaisijat ratkomaan todellisen maailman haku- ja optimointiongelmia. Nykyaikaiset SAT-ratkaisijat antavat kaavan toteutuvuuden

vastauksen lisäksi ratkaisun, joilla arvoilla kaava on toteutuva. Ei toteutuville kaavoille SAT-ratkaisijat palauttavat myös todisteen toteutumattomuudesta. Tällaisia ratkaisijoita kutsutaan täydellisiksi ratkaisijoiksi (*complete solvers*). CNF-kaavan F-malli (model of F) on sijoitukset (*assignment*), joilla kaava on toteutuva [Järvisalo, 2016].

SMT on laajennus SAT-ratkaisijoille. Perustana on 1960-luvulla kehitetty DPLL-algoritmi. Ensimmäisen kertaluvun teoriat (*theories*) voidaan esittää aksiomaattisesti tai ensimmäisen kertaluvun rakenteiden luokkana (*class of first-order structures*) [Moura et al., 2007]. Boolean-kaavojen toteutuvuutta tärkeämpiä SMT-käyttökohteita ovat mallitarkastajat ja teoreematodistajat (*theorem provers*).

Ongelmien enkoodaus SAT-ongelmiksi vaatii ihmisen suorittamaa ajattelua. Hyvän esimerkin enkoodauksesta tarjoaa Lyncen ja Ouakninen [2006] Sudoku-pelin enkoodaus.

2.5. Delta-korjaus

Delta-korjauksen (*delta debugging*) lähtökohtana on ollut testitapausten testisyötteiden minimoiminen niin yksinkertaisiksi kuin mahdollista, jotta testitapaus vielä raportoi testin epäonnistumisesta. Delta-korjauksessa ideana on automaattisesti toistettavien testitapausten syötteiden minimointi automaattisesti ddmin-algoritmilla (*minimizing delta debugging algorithm*). Delta-korjaus on ddmin-algoritmin yleistys [Zeller and Hildebrandt, 2002].

Delta-korjaus ei ole rajoitettu vain ohjelmien syötteisiin, vaan sitä voidaan soveltaa kaikkiin olosuhteisiin, jotka vaikuttavat jollakin tavalla ohjelman suoritukseen. Olosuhteilla tarkoitetaan tässä tapauksessa ohjelman koodia, dataa tietovarastoista (*storage*) tai syötelaitteista, ohjelman ajoympäristöstä ja käytetystä laitteistosta ja niin edelleen [Zeller and Hildebrandt, 2002].

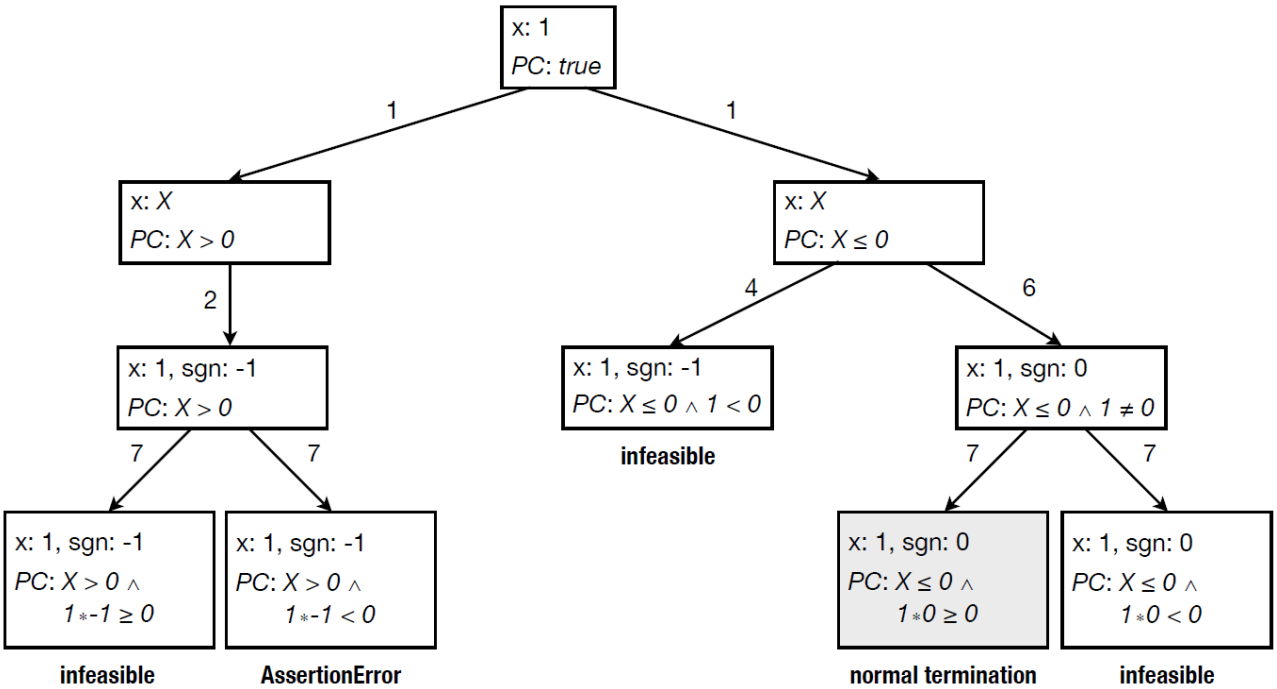
Algoritmin selityksessä käytän alkuperäisestä artikkelista poikkeavia merkintöjä. Lisäksi en esittele formaalia matemaattista esitystä, joka löytyy alkuperäisestä lähteestä. Merkitään R_{pass} tarkoittamaan ohjelman läpäisevää testiajoa ja R_{fail} epäonnistunutta testiä. Olkoon D testien R_{pass} ja R_{fail} ajoihin liittyvien olosuhteiden muutoksien kuvaus, jolle pätee merkintä $D(R_{pass})=R_{fail}$. Tarkka D:n määritelmä riippuu käsiteltävästä ongelmasta ja sen olosuhteista. Tarkemmin D on olosuhteiden muutoksien joukko, joka voidaan pilkkoa pienemmiksi perusmuutoksiksi $D_1 \ldots D_n$. Perusmuutos on myös ongelmakohtainen. Esimerkiksi jos testataan syötteenä annetulla tekstitiedostolla, voi perusmuutos olla yksi merkki. HTML-tiedoston tapauksessa se voi olla esimerkiksi tagi [Zeller and Hildebrandt, 2002].

Jotta testien muutosjoukot olisivat yksiselitteisiä, täytyy käytettäviä muutoksia D_i lajitella jollakin sääntöjenmukaisella tavalla. Merkittävien muutoksien minimointi alkaa binäärihaulla. Alussa muutoksien joukko jaetaan kahdeksi osajoukoksi M_1 ja M_2. Sitten testitapaus ajetaan uudestaan käyttäen erikseen muutoksia M_1 ja uudestaan muutoksilla M_2. Näin saadaan selville mikä puolikas muutoksista aiheuttaa muutoksen testin tulokseen ja prosessia voidaan jatkaa jakamalla muutoksia aiheuttanut puolikas taas kahdeksi pienemmäksi joukoksi. Ongelmallinen tapaus on se, kun muutokset M_1 ja M_2 tuottavat saman tuloksen. Tällöin ruvetaan kasvattamaan muutoksien

osajoukkojen lukumäärää $M_1 \ldots M_n$ ja lopuksi päädytään suurimpaan mahdolliseen osajoukkoon, jolla muutoksia tulee esiin [Zeller and Hildebrandt, 2002].

2.6. Symbolinen suorittaminen

Symbolinen suorittaminen (*symbolic execution*) käsittelee kaikkia syöttömuuttujia symbolisina. Jokaisen muuttujan tila symbolisen suorittamisen aikana esitetään symbolisten syötteiden ilmaisuina. Kun ohjelma haarautuu, jos molemmat valinnat ovat mahdollisia, molemmat polut suoritetaan symbolisella suorittamisella. Jokaisesta suoritetusta polusta kerätään tieto polun tilasta. Polkujen tilat ovat predikaatteja ohjelman syötteille, ja jokainen syöte etenee polkuja, joiden vaatimukset syöte täyttää. Symbolisen suorittamisen tuloste on mahdollisten polkujen ja vastaavien polkujen tilojen joukko. Jokaisesta polusta saadaan symbolisesti esitetty tuloste vastaaviin ohjelman syötteisiin [Nguyen et al., 2013].



Kuva 1. Tilakaavio symbolisesta suorittamisesta taulukossa 3 esitellylle ohjelmalle. Symbolinen suoritus alkaa ylimmästä laatikosta, jossa todellinen muuttujan x arvo korvataan symbolisella arvolla X. Viivojen vieressä olevat numerot merkitsevät, mille riville ohjelmassa siirrytään.

Esimerkkinä käytän taulukon 3 ohjelmaa. Ohjelman symbolisen suorituksen tilakaavio on kuvassa 1, jossa symbolinen suoritus alkaa, kun muuttujaan x asetetaan arvo 1. Nyt muuttujan x arvo korvataan symbolisella arvolla X. Tätä symbolista arvoa käytetään polun tilan PC (Path Condition) tilan ilmaisemisessa sekä muuttujan x arvona riveillä 2 ja 4. Muilla riveillä käytetään muuttujan x todellista arvoa [Torlak et al., 2011].

	`x = 1;`
1	`if (x > 0)`
2	`  sgn = -1;`
3	`else if (x < 0)`
4	`  sgn = -1;`
5	`else`
6	`  sgn = 0;`
7	`assert x * sgn >= 0;`

Taulukko 3. Esimerkkiohjelma symbolisen suorituksen tilakaaviolle joka on esitetty kuvassa 1.

Aina kun PC saa epätosi arvon tai polku päättyy virheeseen, peruutetaan kyseisestä suorituspuun haarasta takaisin ja valitaan seuraava mahdollinen suorittamaton haara. Esimerkkinä virheestä on kuvan 1 `AssertionError`, jossa PC:n kaava on kuitenkin tosi. Kaavat, joilla PC:n arvot lasketaan, riippuvat kyseisen polun ehtolauseista. Tarkempi ja enemmän yksityiskohtiin keskittyvä kirjoitus symbolisesta suorittamisesta löytyy enkeli-debuggauksesta kertovasta artikkelista [Torlak et al., 2011]. Myös Kähkösen [2009] raportti voi valaista ongelmallisia kohtia.

2.7. Enkeli-debuggaus

Enkeli-debuggaus (*angelic debugging*) on menetelmä, joka yhdessä symbolisen suorittamisen kanssa etsii yhden ilmaisun korjausvaihtoehtoja epäonnistuneen testin korjaamiseksi. Tässä kohdassa kerron menetelmästä suuripiirteisesti ja käsittelen tarkemmin vain sitä osaa, jota käytetään myös SemFix-virheenkorjaus työkalussa, josta kerron tarkemmin kohdassa 5.2. Menetelmä etsii korjausehdotusta mahdollisesti vialliselle ilmaisulle *e*, minkä vuoksi jokin testi epäonnistuu. Etsintä alkaa ilmaisun *e* sisältämän arvon *v* muuttamisella sellaiseksi arvoksi *v'*, että testi onnistuu. Jos tällainen arvo *v'* löytyy, lähdetään oletuksesta, että löytyy myös arvon *v'* antama ilmaisu *e'*. Mikäli arvoa *v'* ei löydy, ei ilmaisua *e* pidetä enää viallisena ja siirrytään toiseen ilmaisuun [Torlak et al., 2011].

Menetelmän arvo ohjelmoijalle on sellaisten muutosten esittäminen, jotka eivät missään tapauksessa toimi, ja listata mahdollisia toimivia korjauksia. Menetelmä ei siis suorita itse virheenkorjausta, vaan se jätetään ohjelmoijan tehtäväksi [Torlak et al., 2011].

Menetelmä käyttää enkelisuoritusta (*angelic execution*), jossa valittu ohjelman *P* ilmaisu *e* tulkitaan kyselyksi enkelille (*angel*), joka palauttaa arvon jolla ohjelma suorittaisi testin onnistuneesti, mikäli sellainen arvo on olemassa. Merkitään $P[\alpha/e]$ saaduksi ohjelmaksi, kun *e* ohjelmassa *P* on korvattu symbolisella muuttujalla α. Ilmaisulle *e* ja syötteelle *I* on määritelty enkelitesti (*angelicTest*) seuraavasti:

$AngelicTest(P,I,e) = \exists\alpha.Test(P[\alpha/e],I)$.

Ilmaisu *Test*(*P*,*I*) on tosi jos tulkitsemalla *P* syötteellä *I* antaa odotetun tuloksen. Odotettu tulos on syötteenä annetun *I*:n mukana [Torlak et al., 2011].

Korjausehdotus on ilmaisu, joka korvattuna toisella ilmaisulla muuttaa epäonnistuneen ajon onnistuneeksi. Alustava joukko korjausehdotuksia on määritelty seuraavasti:

$\{e \mid AngelicTest(P,I_f,e)\}$

Tämä joukko saadaan, kun enkelisuoritus tehdään jokaiselle ilmaukselle arvoitettuna sillä polulla, joka suoritetaan epäonnistuneella testillä, joka saadaan syötteellä I_f. Siis jokainen ilmaisu epäonnistuneen testin polulla on enkelimäisen arvioinnin alaisena, ja ne ovat siten korjausehdotuksia. Todellinen arvo jokaiselle korjausehdotukselle, jotka annetulla testisyötteellä läpäisisivät testin, palautetaan *AngelicTest*–kutsulla [Torlak et al., 2011].

Kun alustava korjausehdotusten joukko on saatu, sitä testataan aiemmin läpäistyillä testeillä. Tällä operaatiolla etsitään vastausta kysymykseen, suorittaako ohjelma edelleen hyväksytysti testin, jos korjausehdotuksen *e* arvo vaihdetaan muuksi kuin se normaalisti olisi läpäisevällä syötteellä I_p. Jos ohjelma ei muutetulla arvolla läpäise testiä, niin korjausehdotus *e* poistetaan korjausehdotuskandidaattien joukosta. Tähän kysymykseen vastataan enkelisuorituksen joustavuus tarkistuksella (*flexibility check*):

$FlexTest(P,I,e) = \exists\alpha.(Test(P[\alpha/e],I) \wedge \alpha \neq Eval(P,I,e))$.

Joustavuustarkistuksessa $Eval(P,I,e)$ on arvo, joka saadaan arvoittamalla ilmaisua *e* tulkittaessa ohjelmaa *P* syötteellä *I*. Jos ilmaisua *e* ei suoriteta syötteellä *I*, on $Eval(P,I,e)$ määrittelemätön (*undefined*) ja *FlexTest* tosi [Torlak et al., 2011].

Tätä läpäisevien testien hyödyntämistä SemFix ohjelman kehittäjät kutsuvat lauseketason spesifikaatiopäättelyksi (*statement-level specification inference*) [Nguyen et al., 2013].

Epäonnistuneilla syötteillä I_f ja läpäisevillä syötteillä I_p saadaan korjausehdokkaiden joukko $\{e \mid AngelicTest(P,I_f,e) \wedge \forall\in i\ [1..k].FlexTest(P, I_{P_i},e)\}$ [Torlak et al., 2011].

Joustavuustarkastuksen mielekkyys (*rationale*). Jos ilmaisu *e* on oikea korjauskandidaatti, pitäisi sen olla vaihdettavissa jollakin vielä tuntemattomalla ilmaisulla *e'*. Ilmaisu *e* ei ole oikea kandidaatti, jos sen vaihtaminen mihin tahansa ilmaisuun *e'* aiheuttaa jonkin läpäisevän testin epäonnistumisen. Nyt halutaan käyttää hyväksi läpäiseviä testejä tietämättä ilmaisua *e'* [Torlak et al., 2011].

Ideaalinen kysymys on, onko testitapaus *T* edelleen läpäisevä, vaikka ilmaisu *e* on korvattu toisella ilmaisulla *e'*. Ideaalinen kysymys on liian vaativa vastattavaksi, joten *FlexTest* suorittaa tarkistuksen vastattavalle arviolle. Arvioitava kysymys arvoista on seuraavanlainen:

Oletetaan, että *e* tuottaa arvon *w*, kun testitapaus *T* ajetaan. Onko mahdollista, että *T* on edelleen läpäisevä, jos e korvataan jollakin *w'*, missä $w \neq w'$? Jos *T* ei suorita ilmaisua *e*, niin tarkistus palauttaa arvon tosi [Torlak et al., 2011].

Jos vastaus arvioituun kysymykseen on kyllä, niin silloin vastaus ideaaliseen kysymykseen on kyllä. Jos vastaus arvioituun kysymykseen on ei, niin silloin vastaus ideaaliseen kysymykseen on todennäköisesti ei (*propably no*), mutta vastaus ei ole taatusti ei. Esimerkkinä on lause `t=x+y`. Oletetaan että *e* on `y`, *e'* on 1 ja `t` täytyy olla `x+1` suorituksen myöhemmässä vaiheessa. Tällöin läpäisevässä testitapauksessa, jossa `y` sattuu olemaan täsmälleen 1 ilmaisun *e* kohdalla koodia, arvioitu tarkistus väittäisi korjausehdotuksen olevan ei joustava (*inflexible*). Ideaalinen tarkistus olisi hyväksynyt ilmaisun *e'* [Torlak et al., 2011].

Ongelma on vältetty tekemällä arvoittavan tarkastuksen binäärisestä ominaisuudesta kvalitatiivinen ottamalla huomioon monia läpäiseviä testejä yhdessä samalla kertaa. Kaikesta huolimatta menetelmällä on rajoituksia, jotka johtuvat käytetyistä työkaluohjelmista ja myös itse toteutuksesta. Yksityiskohdat kannattaa lukea lähteestä [Torlak et al., 2011].

2.8. Evolutiivinen laskenta

Evolutiivinen algoritmi on nimitys algoritmeille, jotka jollain tavoin muistuttavat luonnon evoluutiota. Evolutiivisista algoritmeista suurimman ja tunnetuimman ryhmän muodostavat geneettiset algoritmit. Evolutiivisen algoritmin käyttämää tietorakennetta kutsutaan genomiksi. Evolutiivinen algoritmi on evoluutioon perustuva satunnaistettu etsintäalgoritmi. Se ylläpitää ratkaisuehdotusten joukkoa eli populaatiota. Ratkaisuehdotuksista luodaan uusi sukupolvi geneettisten operaatioiden avulla. Näitä ovat esimerkiksi mutaatio ja risteytyminen. Evolutiiviset algoritmit ovat osa heikkojen etsintäalgoritmien perhettä. “Heikko” on yleisnimi laajasti sovellettaville algoritmeille. Heikon etsintäalgoritmin etu on, ettei se tarvitse ongelmasta etukäteisinformaatiota [Salakoski, 2002].

Hyvyys-, sopivuus-, kustannus-, kelpoisuusfunktion (fitness function) tarkoituksena on luokitella ongelmaa varten kehitettyjä ratkaisuehdotuksia. Tavoitteena on etsiä paras ratkaisu ongelmaan. Ratkaisua etsitään tiettyjen ratkaisuehdotuksia luokittelevien kriteerien perusteella, jotka on määriteltävä ongelmakohtaisesti [Salakoski, 2002].

Toinen evolutiivisen laskennan muunnelma, geneettinen ohjelmointi, käyttää ratkaisuehdotuksen esittämiseen ongelmalle luonteenomaista tietorakennetta. Tällöin joudutaan määrittelemään tietorakenteelle soveltuvat geneettiset operaatiot. Ongelmalle räätälöidyt geneettiset operaatiot tekevät algoritmista vahvan ja algoritmin yleiskäyttöisyys kärsii [Salakoski, 2002].

2.9. Ohjelmapaikkaus

Ohjelmapaikkaus (*program patch*) on muutos alkuperäiseen ohjelmaan. Useimmiten paikka lisätään lähdekoodiin, jolloin saatetaan tarvita ohjelman uudelleen kääntämistä. Paikkaus ajetaan yleensä erillisen asennusohjelman kautta, joka tulkitsee paikkauksen ohjeet. Itse paikkaus pitää sisällään ohjeet ja tiedon siitä, mitä pitää ottaa pois paikattavasta ohjelmasta tai lähdekoodista, ja mitä siihen pitää lisätä. Näitä operaatioita varten on myös paikkatieto kohdasta, mihin korjattavasta ohjelmasta poistot ja lisäykset on tehtävä.

2.10. Yksittäinen staattinen sijoitus

Yksittäistä staattista sijoitusta SSA (*static single assingment*) käytetään kääntäjän välikielessä (*intermediate code*). Muuttujan voi alustaa, mutta siihen ei voi sijoittaa. Siitä seuraa, että lähdekielen muuttujasta X luodaan useita versiota (X_1, X_2, …;) ja uusi luodaan aina kun lähdekielessä on sijoitus kyseiseen muuttujaan. Asiaa selventävä esimerkki sivulta https://gcc.gnu.org/onlinedocs/gccint/SSA.html on esitelty muunneltuna taulukossa 4.

	Välikielinen esitys SSA muotoa käyttäen	Alkuperäinen lähdekoodi
1	`if (...)`	`if (...)`
2	`  a_1 = 5;`	`  a = 5;`
3	`else if (...)`	`else if (...)`
4	`  a_2 = 2;`	`  a = 2;`
5	`else`	`else`
6	`  a_3 = 13;`	`  a = 13;`
7		
8	`# a_4 = PHI <a_1, a_2, a_3>`	`//Ei lähdekoodissa`
9	`return a_4;`	`return a;`

Taulukko 4. Esimerkki miten ohjelmakoodin muuttujia käytetään SSA muodossa.

Esimerkissä käytetään uusien muuttujien lisäämisen lisäksi `PHI`-funktiota. Kyseistä funktiota tarvitaan haarautuvassa koodissa, jossa on sijoituksia samaan muuttujaan ja jonka arvoa sitten käytetään haarautuneen koodin jälkeen. Funktio palauttaa yksinkertaistetusti ”jonkin arvon joukosta”. Lisää SSA-muodosta löytyy lähteestä [Cytron et al., 1989].

3. Komponenttipohjainen ohjelman synteesi

Tässä luvussa kerron suppeasti komponenttipohjaisesta ohjelman syntetisoinnista (*component-based program synthesis*). Asian ymmärtämisen helpottamiseksi olen lisännyt taulukon 6, joka toivottavasti selkeyttää mistä on kyse.

Edetään `Brahma`-ohjelman kehittäjien antaman esimerkin avulla. Kyseessä on bittivektori ohjelma, joka muuttaa syötteenä annetun luvun oikeanpuolimmaisimman bittiarvon 1 bittiarvoksi 0. Olkoon `I` ohjelmalle annettu bittivektori. Olkoon `O` ohjelman antama tuloste-bittivektori. Syntetisoinnin käytössä ovat komponentit &-bittioperaattori ja yhdellä vähentäminen eli dekrementointi, jolloin ohjelman koodi olisi `I&(I-1)`. Eräs tapa saada selville tämä vähennyksen ja &-bittioperaattorin yhdistelmä olisi käyttää kaikkien vaihtoehtojen tyhjentävää luettelointia (*enumeration*) [Gulwani et al., 2010].

Olkoon f_1 yksiargumenttinen (*unary*) komponentti, joka toteuttaa `dekrementoi`-operaation. Olkoon f_2 binäärisen `and`-operaation toteutus. Oletetaan, että tiedetään halutun toiminnallisuuden toteutuvan jollakin näiden komponenttien yhdistelmällä. Mahdolliset yhdistelmät on listattu taulukkoon 5, jonka ohjelmista sarakkeiden e ja f ohjelmat toteuttavat halutun toiminnallisuuden. Yhdistelmä saataisiin selville käyttämällä luettelointia ja SMT-ratkaisijaa, mutta käytännössä tämä menetelmä vaatii liikaa aikaa. Sitä vastoin enkoodataan (*encode*) kaikkien kuuden komponenteista f_1 ja f_2 koostuvan silmukattoman (*straight-line*) ohjelman avaruus käyttämällä loogista kaavaa ψ_{wfp}. Tämä kaava on hyvinmuodostuneisuusrajoite ja siihen palataan myöhemmin. Kaava ψ_{wfp} käyttää viittä kokonaislukumuuttujaa, paikkamuuttujaa, josta käytetään merkintää l_x. Kaikille komponenttien syötteille ja tulosteille tarvitaan oma paikkamuuttujansa. Komponenttien tulosteita vastaavien paikkamuuttujien arvot osoittavat rivinumeroa, jossa komponenttia käytetään. Komponentin syötteen osalta rivinumero tarkoittaa, millä rivillä kyseinen syöte on määritelty. Menetelmän etu on lineaarisesti kasvava hakuavaruus tuntemattomien määrään suhteutettuna verrattuna luettelointia käytettäessä tapahtuvaan eksponentiaaliseen hakuavaruuden kasvamiseen [Gulwani et al., 2010].

f-impl (I): 1 O2=f2(I, I); 2 O1=f1(O2); return O1;	f-impl (I): 1 O2=f2(I, I); 2 O1=f1(I); return O1;	f-impl (I): 1 O1=f1(I); 2 O2=f2(I, I); return O2;	f-impl (I): 1 O1=f1(I); 2 O2=f2(O1, O1); return O2;	f-impl (I): 1 O1=f1(I); 2 O2=f2(O1, I); return O2;	f-impl (I): 1 O1=f1(I); 2 O2=f2(I, O1); return O2;
l_{I1}=1 l_{O1}=2 l_{I2}=0 l_{O2}=1 $l_{I'2}$=0	l_{I1}=0 l_{O1}=2 l_{I2}=0 l_{O2}=1 $l_{I'2}$=0	l_{I1}=0 l_{O1}=1 l_{I2}=0 l_{O2}=2 $l_{I'2}$=0	l_{I1}=0 l_{O1}=1 l_{I2}=1 l_{O2}=2 $l_{I'2}$=1	l_{I1}=0 l_{O1}=1 l_{I2}=1 l_{O2}=2 $l_{I'2}$=0	l_{I1}=0 l_{O1}=1 l_{I2}=0 l_{O2}=2 $l_{I'2}$=1
a	b	c	d	e	f

Taulukko 5. Mahdolliset funktion toteutukset, kun funktio koostuu kahdesta operaatiosta f1(dekrementointi) ja f2(&-operaatio). Arvot $l_{I'2}$ ja l_{I2} ilmaisevat, mitkä rivit toimivat syötteinä f2-operaatiolle. Rivi 0 syötteenä tarkoittaa funktiolle annettua parametria. Rivi 1 syötteenä tarkoittaa rivillä 1 olevan operaation antamaa tulosta. Funktion paluuarvo on aina rivillä 2. Kohdissa b ja c ei käytetä rivillä 1 olevaa tulosta eli rivin 1 komponentti on turha. Taulukossa ei myös ole niitä vaihtoehtoja, jolloin kaikki komponentit ovat turhia [Gulwani et al., 2010].

Luodut ohjelmat eivät välttämättä ole haarautumattomia, koska komponenttikirjaston komponentit voivat olla haarautuvia `if-then-else`-funktioita. Siten syntetisoitu funktio voi olla mielivaltainen silmukaton ohjelma [Jha et al., 2010].

Paikkamuuttujien lukumäärä eli paikkamuuttujien joukon suuruus riippuu syntetisointiin käytettävien komponenttien syötteiden lukumäärästä ja komponenttien, eli komponenttien tulosteiden lukumäärästä. Paikkamuuttujien arvoalue, tai lukualue toisin ilmaistuna, riippuu funktiolle annettavien parametrien määrästä ja komponenttien lukumäärästä. Syntetisoitavien rivien lukumäärä on sama kuin komponenttien lukumäärä ja sille välille on rajattu myös komponenttien tulosteiden paikkamuuttujien arvoalue. Nämä paikkamuuttujien arvot määrittävät komponenttien välisen yhteyden.

Tässä esitelty algoritmi on rajoitteihin perustuva (*constraint-based*) lähestymistapa, joka sisältää syntetisointiongelman (*synthesis problem*) pelkistämisen (reducing) rajoitteen ratkaisemiseksi. Pelkistäminen sisältää kaksi avainvaihetta, rajoitteen luominen (*constraint generation*) ja rajoitteen ratkaiseminen (*constraint solving*) [Gulwani et al., 2010].

Rajoitteen luomisvaiheessa syntetisointiongelma enkoodataan syntetisointirajoiteeksi (*synthesis constraint*). Syntetisointirajoitteessa on kaksi mielenkiintoista kohtaa. Syntetisointirajoite on ensimmäisen kertaluvun logiikanyhtälö (*first-order logic formula*). Syntetisointiongelmaa voidaan tarkastella tarkistusongelman (*verification problem*) yleistyksenä. Tiedetään, että silmukattoman ohjelman tarkistaminen voidaan pelkistää ensimmäisen kertaluvun logiikan yhtälön kelpoisuuden (*validity*) todistamiseksi, ja siten syntetisointiongelma voidaan pelkistää toisen kertaluvun logiikkayhtälön toteutuvuuden (*satisfiability*) selvittämiseen. Syntetisointirajoitteen enkoodaus on kuitenkin sellainen, että toisen kertaluvun sijaan se luo ensimmäisen kertaluvun logiikkakaavan, joka on ratkaistavissa valmiiksi saatavilla olevilla rajoitteiden ratkaisijoilla. Lisäksi käytetyn enkoodauksen ansiosta syntetisointirajoitteen koko on taattu olevan rajoitettu komponenttien lukumäärän neliöön [Gulwani et al., 2010].

Rajoitteen ratkaisemisvaiheessa käytetään parannettua muotoa klassisesta vastaesimerkki-ohjatusta iteroivasta syntetisointitekniikasta. Saatu syntetisointirajoite on ∃∀-kaava, jota ei voi suoraan ratkaista saatavilla olevilla SMT-ratkaisijoilla. Vastaesimerkki-ohjattu iteroiva syntetisointitekniikka sisältää aluksi testiarvojen joukon valitsemista universaalikvanttorilla (∀) sidotuille muuttujille ja sitten tuloksena saadun rajoitteen ratkaisemista eksistenssikvanttorilla (∃) sidotuille muuttujille käyttämällä SMT-ratkaisijoita. Jos ratkaisu eksistenssikvanttorilla sidotuille muuttujille toimii myös kaikilla vaihtoehdoilla universaalikvanttorilla sidotuille muuttujille, niin ratkaisu on löydetty. Muussa tapauksessa vastaesimerkki on saatu selville, ja prosessi toistetaan, kun vastaesimerkki on lisätty testiarvojen joukkoon universaalikvanttorilla sidotuilla muuttujilla. Menetelmä toimii loistavasti joillekin rajoitteille, mutta ei syntetisointirajoitteelle [Gulwani et al., 2010].

Parannus iteroivalle syntetisointistrategialle käsittää syntetisointirajoitteen kahden esitystavan käyttämistä. Alkuperäistä ∃∀-esitystapaa käytetään tarkistamaan, onko löydetty ratkaisu eksistenssikvanttorilla sidotuille muuttujille muutamilla testitapauksilla universaalikvanttorilla sidotuilla muuttujilla itseasiassa oikea ratkaisu. Vaihtoehtoista ∃∀∃-esitystapaa, jossa universaalikvanttori kohdistuu vain järjestelmän syötemuuttujiin, käytetään ratkaisun löytämiseen eksistenssikvanttorilla sidotuille muuttujille, joka toimisi supistetulle testitapausten joukolle. Lisätty kvanttori auttaa, koska universaalikvanttori käsittää tällöin vain järjestelmän todelliset syötteet [Gulwani et al., 2010].

Itse syntetisoitava funktio, kuten myös käytettävissä olevat peruskomponentit on määritelty käyttämällä toiminnallista kuvausta (*functional description*). Nämä kuvaukset on annettu loogisen kaavan muodossa, joka yhdistää syötteet ja tulosteet. Ymmärrettävyyden helpottamiseksi yksinkertaistetaan komponentteja siten, että jokaisella komponentilla on vain yksi tuloste. Lisäksi oletetaan, että syötteillä ja tulosteilla on sama tietotyyppi [Gulwani et al., 2010].

Sitä, miten komponenttien funktionaalinen spesifikaatio voidaan enkoodata SMT-ratkaisijan käytettäväksi, ei lähteessä selitetä syvällisesti. Kuitenkin tämä syntetisointimenetelmä toimii vain, jos annettujen komponenttien spesifikaatiot ovat pysähtyviä (*terminating*), kuten pysähtyvässä Turingin koneessa [Gulwani et al., 2010].

Syntetisointiongelma vaatii ohjelman spesifikaation

$$\langle \vec{I}, O, \phi_{spec}(\vec{I}, O) \rangle.$$

Ohjelman spesifikaatio sisältää syötemuuttujien monikkotietotyypin (*tuple*) ja tulostemuuttujan. Spesifikaatiossa on myös lauseke (*expression*) $\phi_{spec}(\vec{I}, O)$, joka määrittelee halutun syöte-tuloste -yhteyden [Gulwani et al., 2010].

Syntetisointiongelmaan tarvitaan myös käytettävien komponenttien spesifikaatioiden joukko:

$$\{\langle \vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i) \rangle | i = 1, \dots, N\}.$$

Tätä spesifikaatioiden joukkoa kutsutaan kirjastoksi, jossa $\phi_i(\vec{I}_i, O_i)$ on spesifikaatio peruskomponentille f_i. Kaikki muuttujat $\vec{I}_i, O_i$ on oletettu erillisiksi. Syntetisointiongelman tavoite on löytää ohjelma `f_impl`, joka toteuttaa oikein ohjelman spesifikaation ϕ_{spec} käyttäen vain

annetun kirjaston komponentteja. Ohjelma `f_impl` saa syötteeksi $\vec{I}$ ja käyttää joukkoa $\{O_1, \dots, O_N\}$ väliaikaisina muuttujina seuraavasti:

$\underline{f\_impl(\vec{I})}$:
$O_{\pi_1} := f_{\pi_1}(\vec{V}_{\pi_1})$;
…;
$O_{\pi_N} := f_{\pi_N}(\vec{V}_{\pi_N})$;
`return` O_{π_N};

jossa $\vec{V}_{\pi_i}$ on joko syötemuuttuja joukosta $\vec{I}$ tai sellainen väliaikainen muuttuja O_{π_j}, että $j<i$, ja $\pi_1,\dots,\pi_N$ on lukujen 1,…,*N* permutaatio ja tarkistusrajoite (*verification constraint*) on voimassa. Tarkistusrajoite on muotoa

$$\forall \vec{I}, O_1, \dots, O_N : \left(\phi_{\pi_1}(\vec{V}_{\pi_1}, O_{\pi_1}) \wedge \dots \wedge \phi_{\pi_N}(\vec{V}_{\pi_N}, O_{\pi_N})\right) \Rightarrow \phi_{spec}(\vec{I}, O_{\pi_N}).$$

Tarkistusrajoite määrää, että jos O_{π_N} on tuloste syötteelle $\vec{I}$ tehdylle toteutukselle, niin O_{π_N} pitäisi myös olla $\vec{I}$ spesifikaation tuloste. Toisin sanoen toteutuksen pitäisi seurata spesifikaatiosta [Gulwani et al., 2010].

Tarkastellaan bittivektoriesimerkin formaalia spesifikaatiota ϕ_{spec} syötettävän bittivektorin *I* ja tuloksena saadun bittivektorin *O* välille. Tässä esimerkkispesifikaatiossa *b* merkitsee bittien lukumäärää bittivektoreissa ja merkintä *I*[*j*] tarkoittaa *j*:tä bittiä bittivektorissa *I*, kun tarkastellaan bittivektoria taulukkona (*array of bits*). Esimerkin spesifikaatio on muotoa

$$\phi_{spec}(I, O) := \bigwedge_{t=1}^{b} \left(\left(I[t] = 1 \wedge \bigwedge_{j=t+1}^{b} I[j] = 0 \right) \Rightarrow \left(O[t] = 0 \wedge \bigwedge_{j \neq t} O[j] = I[j] \right) \right).$$

Esimerkin komponenttikirjasto sisältää *N*=2 komponenttia. Kirjaston komponenteista f_1 on yksiargumenttinen (*unary*) dekrementointioperaatio, ja sen formaali spesifikaatio ϕ_1 syötteen I_1 ja tulosteen O_1 välille on

$$\phi_1(I_1, O_1) := O_1 = (I_1 - 1).$$

Toinen komponentti f_2 on binäärikomponentti, joka toteuttaa ja-operaation. Formaali spesifikaatio ϕ_2 syötteiden I_2, I'_2 ja tulosteen O_2 välille on

$$\phi_2(I_2, I'_2, O_2) := O_2 = (I_2 \,\&\, I'_2).$$

Tarkistusrajoitteesta saadaan syntetisointirajoitteen kaltainen, kun sen jokainen atominen fakta (*atomic fact*) $\phi_{\pi_i}(\vec{V}_{\pi_i}, O_{\pi_i})$ edellytyksessä (*antecedent*) korvataan ilmauksella $\phi_{\pi_i}(\vec{V}_{\pi_i}, O_{\pi_i}) \wedge \vec{I} = \vec{V}_{\pi_i}$. Samoin seurauksesta (*consequent*) korvataan $\phi_{spec}(\vec{I}, O_{\pi_N})$ ilmauksella $\phi_{spec}(\vec{I}, O)$ edellyttäen, että edellytykseen lisätään $O = O_{\pi_N}$. Siten tarkistusrajoite voidaan kirjoittaa uudestaan muodossa

$$\forall \vec{I}, O, \vec{I}_1, \dots, \vec{I}_n, O_1, \dots, O_N : \left((O = O_{\pi_N}) \wedge \bigwedge_{i=1}^{N} (\phi_i(\vec{I}_i, O_i) \wedge \vec{I}_i = \vec{V}_i) \right) \Rightarrow \phi_{spec}(\vec{I}, O).$$

Tästä tarkistusrajoitteen kaavan edellytyspuoli jaetaan osiksi

$$\phi_{lib} := \left(\bigwedge_{i=1}^{N} \phi_i(\vec{I}_i, O_i) \right)$$

ja liitettävyysrajoite (*connectivity constraint*) osiksi

$$\phi_{conn} \coloneqq \left(O = O_{\pi_N}\right) \wedge \left(\bigwedge_{i=1}^{N} \vec{I}_i = \vec{V}_i\right).$$

Lisäksi peruskomponenttien formaaleista syötteistä ja tulosteista määritellään joukot P ja R seuraavasti:

$$P \coloneqq \bigcup_{i=1}^{N} \vec{I}_i.$$

$$R \coloneqq \bigcup_{i=1}^{N} \{O_i\} = \{O_1, \dots, O_N\}.$$

Täten tarkistusrajoite voidaan kirjoittaa muodossa

$$\forall \vec{I}, O, P, R: \left(\phi_{lib}(P, R) \wedge \phi_{conn}\left(\vec{I}, O, P, R\right)\right) \Rightarrow \phi_{spec}\left(\vec{I}, O\right).$$

Tarkistusrajoitteessa ϕ_{lib} edustaa peruskomponenttien spesifikaatiota ja liitettävyysrajoite ϕ_{conn} edustaa yhteenliittämisiä, jotka sisältävät kuvaukset formaaleista toteutumiin (*actuals*) ja jonkin komponentin paluuarvosta syntetisoitavan ohjelman tulosteelle. Liitettävyysrajoite on samanarvoisuuksien konjunktio joukoissa P ja R olevien muuttujien välillä. Liitettävyysrajoite määrittelee peruskomponenttien järjestyksen syntetisoidussa ohjelmassa ja peruskomponenttien syöteparametrien arvot [Gulwani et al., 2010].

Taulukon 5 ohjelmalla e tarkistusrajoite on kaava

$$\forall I, O, I_1, I_2, I'_2, O_1, O_2\left(\phi_{lib} \wedge \phi_{conn} \Rightarrow \phi_{spec}\right),$$

josta komponenttien kirjasto on

$$\phi_{lib} \coloneqq \ \phi_1(I_1, O_1) \wedge \phi_2(I_2, I'_2, O_2)$$

ja liitettävyysrajoite on

$$\phi_{conn} \coloneqq I_1 = I \wedge I_2 = O_1 \wedge I'_2 = I \wedge O = O_2.$$

Ratkaistavana oleva tarkistusrajoite on universaalisti kvantifioitu kaava. Tarkistusrajoitteen kaavan kelpoisuuden (*validity*) päätettävyyden (*deciding*) monimutkaisuus riippuu valitusta ilmaisukielestä, jolla määritellään ohjelman spesifikaatio ϕ_{spec} ja kirjaston spesifikaatiot ϕ_i. Jos ilmaisukieli on sellaisen kielen alijoukko, jota SMT-ratkaisijat pystyvät käsittelemään, niin silloin voidaan käyttää valmiita SMT-ratkaisijoita ratkaisemiseen ja ratkaista tarkistusongelma (*verification problem*). Toisin sanoen universaalin kaavan kelpoisuus tarkistetaan kysymällä SMT-ratkaisijaa tarkistamaan kaavan, eli tarkistusrajoitteen negaation toteutuvuus [Gulwani et al., 2010].

Syntetisoitavan ohjelman liitettävyysrajoitetta ϕ_{conn} eli kirjaston peruskomponenttien syötteiden ja tulosteiden välisiä yhteenliittymiä ei tiedetä. On myös huomattava, että liitettävyysrajoite on osa tarkistusrajoitetta. Siten syntetisointiongelma vastaa seuraavan rajoitteen ratkaisemista

$$\exists \phi_{conn}: \forall \vec{I}, O, P, R:$$
$$\left(\phi_{lib}(P, R) \wedge \phi_{conn}\left(\vec{I}, O', P, R\right)\right) \Rightarrow \phi_{spec}\left(\vec{I}, O\right).$$

Rajoitteessa on toisen kertaluvun eksistenssikvanttori kaikkien mahdollisten yhteenliittymien joukon yli. Toisen kertaluvun eksistenssikvanttori voidaan muuntaa ensimmäisen kertaluvun

eksistenssikvanttoriksi. Perusidea on lisätä ensimmäisen kertaluvun kokonaislukumuuttujia eli paikkamuuttujia, joiden arvot määrittävät komponenttien väliset yhteenliittymät. Paikkamuuttujat myös määräävät, mille riville syntetisoitavassa ohjelmassa kukin komponentti sijoitetaan, ja miltä riviltä tai mistä ohjelmalle annetuista argumenteista komponentti saa argumenttinsa. Paikkamuuttujien joukon määrittely on muotoa

$$L \coloneqq \{l_x | x \in P \cup R\}.$$

Tähän joukkoon lisätään muuttuja l_x jokaista joukon $P \cup R$ muuttujaa x kohden. Muuttujan l_x tulkinta riippuu muuttujan *x* arvosta. Jos *x* on komponentin f_i tulostemuuttuja O_i, eli joukossa *R*, niin l_x edustaa ohjelman riviä, jossa komponenttia f_i käytetään. Jos x on komponentin f_i *j*:nes syöteparametri, niin l_x:n arvo edustaa paikkaa, josta komponentti f_i saa *j*:nen syöteparametrinsa. Paikka tarkoittaa tässä tapauksessa joko ohjelman riviä tai jotakin ohjelmalle annettua syöteparametria. Tätä arvojen tulkinnan ymmärtämistä voi auttaa taulukko 6. Kaikkia mahdollisia paikkamuuttujien arvoja kuvaa kokonaislukujen joukko $\{0, \dots, M-1\}$, missä M on ohjelman syötteiden lukumäärän $\vec{I}$ ja kirjaston komponenttien lukumäärän *N* summa. Toisin ilmaistuna $M = N + |\vec{I}|$. Näille paikkamuuttujien arvoille voidaan antaa tulkinta: *j*:s syöte löytyy paikasta *j*-1, kun *j* on vähintään 1 ja enintään yhtäsuuri kuin *M*. Arvot, jotka voidaan tulkita *j*:ksi riviksi eli komponentin sijainniksi, saadaan laskemalla *J*+$|\vec{I}|$-1. Taulukon 5 esimerkkiä käyttäen paikkamuuttujien joukko *L* koostuu viidestä kokonaislukumuuttujasta $L = \{l_{O_1}, l_{O_2}, l_{I_1}, l_{I_2}, l_{I_{I_2}}\}$. Muuttujat l_{O_1} ja l_{O_2} osoittavat, missä komponentteja f_1 ja f_2 käytetään. Muuttuja l_{I_1} osoittaa komponentin f_1 syötteen määrittelyn paikan. Muuttujat l_{I_2} ja $l_{I_{I_2}}$ osoittavat komponentin f_2 syötteiden määrittelyjen paikat. Esimerkin komponenttien lukumäärä *N*=2 ja syötteitä on vain yksi, jolloin *M* =3. Muuttujat l_{O_1} ja l_{O_2} saavat arvonsa joukosta $\{1,2\}$ ja muuttujat $l_{I_1}, l_{I_2}, l_{I_{I_2}}$ joukosta $\{0,1,2\}$ [Gulwani et al., 2010].

Paikka-muuttuja l_x	Funktion syötteelle ei paikka-muuttujaa	1	2	3	4	5
Toiminto	Funktion syötepara-metri	AND-operendin 1. syötepara-metrin osoitin	AND-operendin 2. syötepara-metrin osoitin	Dekrementoi yhdellä operendin syötepara-metrin osoitin	AND-operen-din tulos-teen osoitin	Dekremen-toi yhdellä operendin tulosteen osoitin
Arvo eli ohjelman rivit eli paikka	0	0	1	0	2	1

Taulukko 6. Eräs mahdollinen paikkamuuttujien järjestys taulukon 5 kohdan f ohjelmalle. AND-operendin laitoin tietoisesti paikkamuuttujien tulkintaan ennen dekrementointioperendia havainnollistamaan että komponenttien järjestyksellä paikkamuuttujissa ei ole mitään väliä. Ainoastaan paikkamuuttujien arvoilla ja paikkamuuttujien tulkinnalla on merkitystä.

Joukon L muuttujille annetut arvot eivät välttämättä takaa hyvin muodostunutta ohjelmaa (*well-formed program*). Kun joukon L muuttujat toteuttavat johdonmukaisuusrajoiteen ψ_{cons} (*consistency constraint*) ja asyklisyysrajoiteen ψ_{acyc} (*acyclicity constraint*), syntetisoitu ohjelma on hyvin muodostunut. Johdonmukaisuusrajoite ψ_{cons} ja asyklisyysrajoite ψ_{acyc} ovat osa hyvinmuodostuneisuusrajoitetta ψ_{wfp} (*well-formedness constaint*). Hyvinmuodostuneisuusrajoitteen määrittely:

$$\psi_{wfp}(L) \coloneqq \bigwedge_{x \in P} (0 \leq l_x < M - 1) \wedge \bigwedge_{x \in R} \left(|\vec{I}| \leq l_x < M - 1\right) \wedge \psi_{cons}(L) \wedge \psi_{acyc}(L).$$

Hyvinmuodostuneisuusrajoitteeseen sisältyvä johdonmukaisuusrajoite $\psi_{cons}(L)$ rajoittaa yhdelle syntetisoitavalle riville tulevien komponenttien määrän yhteen. Rajoitteen määrittely perustuu käytettyyn enkoodaukseen, jossa l_{O_i} enkoodaa rivinumeron, jossa komponenttia f_i käytetään. Komponentit sijoittuvat omille riveilleen, jos eri i:n arvoille tulee eri l_{O_i}:n arvot. Johdonmukaisuusrajoite koskee siis vain komponenttien tulostetta ja siten johdonmukaisuusrajoitteen määrittelyssä esiintyvät muuttujat x ja y kuuluvat joukkoon R (*Return variables*). Johdonmukaisuusrajoitteen määrittely on muotoa

$$\psi_{cons}(L) \coloneqq \bigwedge_{x,y \in R, x \neq y} (l_x \neq l_y).$$

Asyklisyysrajoite ψ_{acyc} tarkoittaa, että jokaisella komponentilla, jos x on syöte komponentille ja y on komponentin tuloste, niin paikka l_x jossa syöte on määritelty, pitää esiintyä ennen kuin paikka

l_y jossa komponenttia käytetään ja sen tuloste on määritelty. Asyklisyysrajoiteen määrittely on muotoa

$$\psi_{acyc} \coloneqq \bigwedge_{i=1}^{N} \bigwedge_{x \in \vec{X}_i, y \equiv O_i} (l_x < l_y).$$

Hyvin muodostetussa ohjelmassa muuttujat on alustettu ennen niiden käyttöä. Käytetyssä enkoodauksessa komponenttia f_i käytetään paikassa l_{O_i} ja sen syötteet tulevat joukossa $\{l_x | x \in \vec{I}_i\}$ osoitetuista paikoista. On myös huomattava, että hyvinmuodostuneisuusrajoitteen enkoodaus sisältää myös paikkamuuttujien tulkinnan [Gulwani et al., 2010].

Mikäli paikkamuuttujat *L* toteuttavat hyvinmuodostuneisuusrajoitteen ψ_{wfp}, niin *L* määrittelee hyvin muodostuneen silmukattoman ohjelman staattisessa yhden sijoituksen (*static single assingment*, SSA, selitetty kohdassa 2.10) muodossa. Nämä sijoitukset tekevät kutsuja kirjaston komponentteihin. Tarkemmin ilmaistuna funktio *Lval2Prog(L)* palauttaa joukon *L* muuttujien annettuja arvoja vastaavan ohjelman. Täten funktiossa *Lval2Prog(L)* tehdään *i*:ttä riviä vastaava sijoitus $O_j := f_j(O_{\sigma(1)}, \dots, O_{\sigma(t)})$, jos $l_{O_j} = i$, $l_{I_j^k} = l_{\sigma(k)}$, jossa *k* saa arvoja välillä 1,…,*t*, kun *t* on komponentin f_j saamien argumenttien lukumäärä eli ariteetti (*arity*), ja $(I_j^1, \dots, I_j^t)$ on komponentin f_j syötteiden $\vec{I}_j$ syöttömuuttujien monikko (*tuble*) [Gulwani et al., 2010].

Käytetyn enkoodauksen ansiosta funktiolla *Lval2Prog(L)* on bijektiivisen kuvauksen ominaisuus. Siten jokaiselle maalijoukon alkiolle on olemassa kuvaus lähtöjoukossa. Bijektiivisyydestä seuraa lisäksi, että maalijoukon alkion kuvauksia on lähtöjoukossa enintään yksi [Jha et al., 2010]. Olkoon joukko **L** kaikkien joukon *L* hyvinmuodostuneisuusrajoitteen toteuttavien arvojen joukko. Olkoon **Π** kaikkien haarautumattomien ohjelmien joukko, jotka on luotu komponenttikirjaston komponenteista. Tämä on kuvaus joukosta **L** joukkoon **Π,** ja se on bijektiivinen. Funktio *Lval2Prog(L)* palauttaa silmukattoman ohjelman staattisessa yhden sijoituksen muodossa [Gulwani et al., 2010].

Esimerkkinä annetussa ohjelmassa ψ_{wfp} saadaan seuraavasti:

$$\psi_{wfp}(L) \coloneqq \psi_{cons}(L) \wedge \psi_{acyc}(L) \wedge \bigwedge_{x \in P} (0 \leq l_x < 2) \wedge \bigwedge_{x \in R} (1 \leq l_x < 2),$$

jossa $\psi_{cons} \coloneqq (l_{O_1} \neq l_{O_2})$, ja $\psi_{acyc} \coloneqq (l_{i_1} < l_{O_1}) \wedge (l_{i_2} < l_{O_2}) \wedge (l_{i_{\prime 2}} < l_{O_2})$.
Esimerkin tapauksessa saadaan joukot $R = \{O_1, O_2\}$ ja $P = \{\vec{I}_1, \vec{I}_2, \vec{I}_{\prime 2}\}$. Rajoitteen ψ_{wfp} toteuttavia ratkaisuja on kuusi kuten on nähtävissä taulukosta 5 [Gulwani et al., 2010].

Liitettävyysrajoite, jolla saadaan liitettyä komponenttien syötteet ja tulosteet syntetisoitavan ohjelman syötteisiin ja tulosteisiin joukon *L* arvoja käyttäen, määritellään seuraavasti:

$$\psi_{conn} \coloneqq \bigwedge_{x,y \in P \cup R \cup \vec{I} \cup \{o\}} (l_x = l_y \Rightarrow x = y).$$

Aiemmin esiteltiin liitettävyysrajoite

$$\phi_{conn} \coloneqq (O = O_{\pi_N}) \wedge \left(\bigwedge_{i=1}^{N} \vec{I}_i = \vec{V}_i \right).$$

Nyt tässä kohdassa esitettyä liitettävyysrajoitetta ψ_{conn} tullaan käyttämään syntetisointirajoitteessa.

Tarkastellaan seuraavaksi ensimmäisen kertaluvun syntetisointirajoitetta, joka enkoodaa syntetisointiongelman. Koska kaikkien kelvollisten ohjelmien joukko voidaan kuvata paikkamuuttujien L arvoilla, voidaan syntetisointiongelma pelkistää sellaisten joukon L arvojen löytämiseen, jotka täyttävät hyvinmuodostuneisuusrajoiteen vaatimukset, ja saatu hyvinmuodostunut ohjelma on oikea tarkistusrajoitteessa kuvatulla tavalla. Näin ollen syntetisointirajoite (*synthesis constraint*) voidaan kirjoittaa muodossa

$$\exists L\colon \Big(\psi_{wfp}(L) \wedge \forall \vec{I}, O, P, R\colon \phi_{lib}(P,R) \wedge \psi_{conn}\big(\vec{I}, O, P, R, L\big) \Rightarrow \phi_{spec}\big(\vec{I}, O\big)\Big).$$

Saadulle rajoitteelle voidaan tehdä seuraavat muutokset. Muuttujien joukot P ja R voidaan yhdistää ja nimetä muuttujista saatu joukko joukoksi T. Poistamalla rajoitteesta universaalikvanttori, voidaan syntetisointirajoite kirjoittaa muodossa

$$\exists L \forall \vec{I}, O, T\colon \psi_{wfp}(L) \wedge \Big(\phi_{lib}(T) \wedge \psi_{conn}\big(\vec{I}, O, T, L\big) \Rightarrow \phi_{spec}\big(\vec{I}, O\big)\Big).$$

Taulukon 5 ohjelmista vain vaihtoehdot e ja f toteuttavat koko syntetisointirajoitteen [Gulwani et al., 2010].

Hahmotellaan vielä, miten syntetisointirajoitteessa voidaan käyttää liitettävyysrajoitteesta muotoa ψ_{conn}. Olkoon syntetisointirajoite ψ, jonka koko on $O(n+m^2)$, kun n on ohjelman spesifikaation ja kirjaston komponenttien spesifikaatioiden koko ja m on kirjaston komponenttien lukumäärä. Paikkamuuttujien L joukon suuruus on taas $O(m)$. Oletetaan, että ψ on kelvollinen. Tästä seuraa että joukolle L on olemassa arvot, merkitään L_0, siten, että $\psi_{\text{wfp}}(L_0)$ pätee ja kaava $\forall \vec{I}, O, P, R\colon \phi_{lib}(P,R) \wedge \psi_{conn}\big(\vec{I}, O, P, R, L_0\big) \Rightarrow \phi_{spec}\big(\vec{I}, O\big)$ on kelvollinen. Koska $\psi_{\text{wfp}}(L_0)$ pätee, voidaan ohjelmaa nimellä P saada funktiolla *Lval2Prog(L)*. Nyt funktion *Lval2Prog* määritelmä yhdessä rajoitteen ψ_{conn} kanssa takaavat, että ohjelman P määrittelemä liitettävyysrajoite ϕ_{conn} ja liitettävyysrajoite $\psi_{conn}(L_0)$ ovat ekvivalentteja. Täten tarkistusrajoite ohjelman P oikeellisuudelle on kelvollinen. `Brahma`-ohjelman kehittäjät myös todistavat, että ohjelmasta P, joka käyttää vain kirjaston komponentteja, voidaan johtaa paikkamuuttujien arvot joukossa L. En kuitenkaan esittele sitä koska oleellista tässä kohdassa on liitettävyysrajoitteen ϕ_{conn} korvaaminen rajoitteella ψ_{conn} [Gulwani et al., 2010].

Syntetisointirajoite, jossa kvanttorit vuorottelevat ∃∀-muodossa, ratkaistaan seuraavasti. Yleistä ratkaisijaa voidaan käyttää modulaarisesti saatavilla olevien toteutuvuusratkaisijoiden kanssa. Yleistä ratkaisijaa voidaan käyttää myös muiden rajoitteiden, eikä vain syntetisointirajoitteen ratkaisemiseen. Se on kuitenkin liian tehoton syntetisointirajoitteen ratkaisemiseen, joten sitä muokataan tehokkaammaksi syntetisointirajoiteen ratkaisemista varten [Gulwani et al., 2010].

1.	// Input $\exists L \forall \vec{I}{:}\phi$ is an exists-forall formula
2.	// Output: unsatisfiable or satisfiable
3.	$S{:=}\{\vec{I}_0\}$ //$\vec{I}_0$ is an arbitrary value for $\vec{I}$
4.	while (1) {
5.	model :=T-SAT($\exists L{:}\wedge_{\vec{I}_0 \in S}\ \phi(L, \vec{I}_0)$);
6.	if(model $\neq \perp$){curr$L$:=model | L }
7.	else{return("unsatisfiable")};
8.	model:=T-SAT($\exists \vec{I}{:}\neg\phi$(curr$L$, $\vec{I}$));
9.	if(model $\neq \perp$){$\vec{I}_1$=model$\|\vec{I}$; $S{:=}S\cup\{\vec{I}_1\}$}
10.	else{return("satisfiable")};
11.	}

Taulukko 7. Proseduurin StandardExAllSolver pseudokoodiesitys.

Yleistä systetisointirajoitteen ratkaisijaa kuitenkin tarvitaan aina ensimmäisen ehdokkaan löytämiseksi. Tämän `StandardExAllSolver` ratkaisijan pseudokoodi on esitetty taulukossa 7. Prosessin alussa valitaan mielivaltaisesti alkuarvot $\vec{I}_0$, eli niin sanottu siemen, syötteinä oleville muuttujille, jotka sijoitetaan muuttujaan *S* rivillä 3. Tämän jälkeen suoritetaan iteratiivisesti seuraavat vaiheet:

Riveillä 5-7 suoritetaan syötteenä annetun kaavan rajoitettu *L*-arvojen haku. Rajoitus tarkoittaa tässä sitä, että annetun kaavan toteuttavat *L*-arvot saadaan joillakin mahdollisilla syötteillä. Jos kaavan toteuttavat arvot löydetään, siirrytään eteenpäin. Jos toteuttavia *L*-arvoja ei löydy yhtään, niin syötteeksi annettu kaava julistetaan toteutumattomaksi.

Riveillä 8-9 tarkistetaan, toimivatko edellisessä vaiheessa saadut *L*-arvot kaikilla mahdollisilla syötteillä. Jos toimivat, niin halutut *L*-arvot ovat löytyneet. Muussa tapauksessa löydetään uudet syötearvot, joilla syötteenä annettu kaava ei toteudu, ja lisätään uudet syötearvot testattavien syötearvojen joukkoon *S* rivillä 9 ja palataan riveille 5-7. Rivillä 8 on huomattava negaatio verrattuna rivin 5 mallin (*model*) luomiseen. Negaatio esiintyy, koska proseduuri on vastaesimerkki-ohjattu (*counterexample-guided*) [Gulwani et al., 2010].

1.	// Input $\exists L \forall \vec{I}, O, T{:}\psi_{wfp} \wedge (\phi_{lib} \wedge \psi_{conn} \Rightarrow \phi_{spec})$ is synthesis constraint
2.	// Output: Synthesis failed or values for *L*
3.	*S*:={$\vec{I}_0$} //$\vec{I}_0$ is an arbitrary input
4.	while (1) {
5.	model := T-SAT ($\exists L, O_1, \dots, O_n, , T_1, \dots, T_n{:}\psi_{wfp}(L) \wedge \wedge_{\vec{I}_i \in S}(\phi_{lib}(T_i) \wedge \psi_{conn}(\vec{I}_i, O_i, T_i, L) \wedge \phi_{spec}(O_i, \vec{I}_i))$);
6.	if(model $\neq \perp$){curr*L*:=model | *L* }
7.	else{return("Synthesis failed")};
8.	model:=T-SAT($\exists \vec{I}, O, T{:}\ \psi_{conn}(\vec{I}, O, T, \text{curr}L) \wedge \phi_{lib}(T) \wedge \neg\phi_{spec}(\vec{I}, O)$) ;
9.	if(model $\neq \perp$){$\vec{I}_1$=model|$\vec{I}$; *S*:=*S*∪ {$\vec{I}_1$}}
10.	else{return(curr*L*)};
11.	}

Taulukko 8. Proseduurin RefinedExAllSolver pseudokoodiesitys.

Taulukossa 8 on muunneltu syntetisointirajoitteen ratkaisija. Taulukoissa 7 ja 8 esiintyvä funktio T-SAT on kutsu SMT-ratkaisijalle. Taulukon 7 proseduurin ongelmana on, että joillakin syötteillä epäonnistunut syntetisointi yritetään vain korvata erilaisella syntetisoinnilla, sen sijaan, että yritettäisiin muokata alkuperäistä syntetisointia sellaiseksi, joka toimisi useammilla erilaisilla syötteillä [Gulwani et al., 2010].

Muunneltu versio `RefinedExAllSolver` käyttää kahta erilaista versiota syntetisointirajoitteesta kahdessa vaiheessa. Kaava (F_{ver}) on sama kuin syntetisointirajoite. Kaava (F_{syn}) on heikompi versio syntetisoinnista

(F_{ver}) $\exists L \forall \vec{I}, O, T : (\psi_{wfp} \wedge(\phi_{lib} \wedge \psi_{conn} \Rightarrow \phi_{spec}))$

(F_{syn}) $\exists L \forall \vec{I}, \exists O, T : (\psi_{wfp} \wedge(\phi_{lib} \wedge \psi_{conn} \wedge \phi_{spec}))$.

Kahdessa vaiheessa toimiva muunneltu proseduuri, joka on esitelty taulukossa 8 on taulukossa 7 esitetyn proseduurin kaltainen. Äärellinen syntetisointivaihe on taulukossa 8 riveillä 5-7. Tässä vaiheessa syntetisoidaan malli, joka toimii rajatulle määrälle monia syötteitä. Proseduuri hakee arvot joukolle *L*, jotka toimivat kaikilla syötteillä joukossa *S* riveillä 5-6. Rivi 5 ratkaisee kaavan (F_{syn}). Tarkistusvaihe suoritetaan riveillä 8-10, jolloin tarkistetaan, että syntetisoitu malli joka toimii syötteille joukossa *S*, toimii myös kaikilla mahdollisilla syötteillä. Eli jos luodut arvot *currL* joukolle *L* toimivat kaikilla syötteillä, haluttu malli on löytynyt. Muutoin ollaan löydetty uudet syötteet $\vec{I}_1$, joilla malli ei toiminut ja jotka lisätään joukkoon *S* rivillä 9. Rivillä 8 tarkistetaan kaava (F_{ver}) joka on siis syntetisointirajoite [Gulwani et al., 2010].

Proseduurit `RefinedExAllSolver` ja `StandardExAllSolver` suorittavat samankaltaisia operaatioita samoilla riveillä. Käyttämällä kaavaa (F_{syn}) syntetisointivaiheessa syntetisoitaessa joukon *L* arvoilla saatavaa mallia, taataan, että malli toimii kaikilla joukossa *S* olevilla syötteillä. Tarkistusvaiheessa jatketaan varsinaisen syntetisointirajoitteen käyttämistä [Gulwani et al., 2010].

CompositionSynthesis($\phi_{spec}, \{\phi_i \mid i = 1, \dots, N\}$):	

// Input: ϕ_{spec}: component specification
// $\{\phi_i \mid i = 1, \dots, N\}$: library specification
// Output: Failure/Program implementing ϕ_{spec}	

1	Let $\exists L \forall \vec{I}, O, P, R : \psi_{wfp} \wedge (\phi_{lib} \wedge \psi_{conn} \Rightarrow \phi_{spec})$ be the synthesis constraint.
2	L := RefinedExAllSolver($\psi_{wfp}, \phi_{lib}, \psi_{conn}, \phi_{spec}$);
3	if (L ≠ "synthesis failed") {return(Lval2Prog(L))}
4	else {return("synthesis failed")};

Taulukko 9. Proseduurin CompositionSynthesis pseudokoodiesitys.

Täydellinen syntetisointiproseduurin pseudokoodiesitys on esitetty taulukossa 9. Rivillä 2 kutsutaan proseduuria `RefinedExAllSolver` joukon L arvojen saamista varten. Koodin syntetisointi tapahtuu rivillä 3 kutsuttaessa proseduuria `Lval2Prog`. Liitettävyysrajoitteessa käytetään versiota ψ_{conn} version ϕ_{conn} sijaan [Gulwani et al., 2010].

Todistuksen, että `RefinedExAllSolver` palauttaa aina oikean vastauksen päättyessään, olen päättänyt jättää kirjoittamatta. Koska tätä menetelmää kuitenkin käytetään, oletan sen olevan toimiva. Todistus löytyy tarvittaessa lähteestä [Gulwani et al., 2010].

4. Testauksesta

Tässä luvussa kerron hieman menetelmistä, jotka liittyvät automaattiseen dynaamiseen testaukseen, koska ne liittyvät automaattisesti toteutettavaan virheiden korjaukseen. Dynaamisessa testauksessa testattava sovellus aina suoritetaan jollakin testisyötteellä ja verrataan ohjelman antamaa tulostetta haluttuun tulokseen. Staattisessa testauksessa tarkastellaan ohjelman lähdekoodeja ja muita ohjelmaan kuuluvia tietoja. Tässä kohdassa ei käsitellä staattista testausta, mutta jotkin käsiteltävät asiat voidaan myös tulkita staattiseksi testaukseksi. Testaustyökaluista lähemmässä tarkastelussa on JUnit kohdassa 4.2, jota käytän esimerkkinä siitä, mitä dynaaminen testaus vaatii ohjelmoinnissa.

Lisäksi kerron formaalista tarkistamisesta kohdassa 4.5. Näiden menetelmien täydellinen käyttäminen voi korvata testaamisen kokonaan. Menetelmät ovat kuitenkin varsin työläitä ja vaativat matemaattista koulutusta enemmän kuin ohjelmointi imperatiivisilla ohjelmointikielillä, joten käytännössä niitä käytetään vain hyvin kriittisten kohtien tarkistamiseen. Lisäksi on olemassa tapauksia, joihin nämäkään menetelmät eivät pysty antamaan täyttä varmuutta, joten pitäisin näitä menetelmiä käytännössä ennemmin täydentävinä kuin korvaavina.

Tarkastuksella varmistetaan että tuote vastaa vaatimuksia ja määritelmiä. Kelpoistaminen (*validation*) on menetelmä, jolla tutkitaan tuotteen sopivuus käyttötarkoitukseensa [Rauhala, 2010].

4.1. Dynaaminen testaus

Tarkastellaan lähemmin sellaisia testausmenetelmiä, jotka liittyvät automaattiseen virheidenkorjaukseen. Konfiguraatiotestausta en käsittele, vaan vain mainitsen sen, koska virheidenkorjaustyökalut voidaan pienellä muutoksella muuttaa toimimaan myös konfiguraatiotiedostoja korjaaviksi. Ongelmallisempaa on konfiguraatiotestauksen syötteiden ja testien tulosten automaattinen hyödyntäminen. Käsittelemistäni virheenkorjaustyökaluista mikään ei sellaisenaan toimi konfiguraatiotestausta hyödyntävästi, mutta tämä on hyvä mainita tulevaisuuden kehitystä varten.

Dynaaminen analyysi tarkoittaa ohjelmiston testaamista ohjelmakoodia suorittamalla. Tällöin keskitytään ohjelman ja koodin käyttäytymiseen ohjelman suorituksen aikana [Rauhala, 2010].

Mustalaatikkotestaus (*black-box testing*) perustuu testattavana olevan kohteen (ohjelma tai funktio) syötteisiin ja tulosteisiin. Mustalaatikkotestauksessa ei välitetä testattavana olevan kohteen rakenteesta tai sisällöstä (koodista), vaan tutkittavana ovat kohteen tulosteet erilaisilla syötearvoilla. Testaajalle kohde on ikään kuin musta laatikko. Testauksen oikeellisuutta tarkastellaan vertaamalla saatuja tuloksia oikeisiin tai haluttuihin tulosteisiin. Ohjelman ja sen ympäristön pitää jäädä suorituksen jälkeen oikeaan tilaan. Testitapaukset johdetaan aina kohteen määrittelyn perusteella [Rauhala, 2010].

Lasilaatikkotestauksessa (*white-box testing*, *glass-box testing*) testaaja näkee ohjelmiston rakenteen eli koodin. Testitapahtumat voidaan suunnitella koodin perusteella. Testaus tapahtuu nimenomaan ohjelmiston toiminnallisuuden perusteella eikä testaaja välitä ohjelmistolle asetetuista vaatimuksista. Lasilaatikkotestausta suoritetaan moduulitestaustasolla, koska testauksessa

keskitytään yksityiskohtiin. Lasilaatikkotestauksessa testaus etenee loogisia polkuja pitkin. Tämän takia testitapausten valinnassa tavoitteena on, että kaikki kohteen (ohjelman tai funktion) haarat ja ohjelmapolut tulisivat käytyä läpi [Rauhala, 2010].

Harmaalaatikkotestaus (*gray box testing*) on musta- ja lasilaatikkotestauksen välimuoto. Tekniikassa käytetään hyväksi tietoa ohjelman toteutusperiaatteista. Testataan yleensä raja-alueita kuten ali- ja ylivuotoja, nollalla jakamista, pyöristyksiä ja esimerkiksi kuukauden alku- ja loppupäiviä [Rauhala, 2010].

Moduulitestauksessa kohteena on vain yksittäinen moduuli. Moduulin toteuttamiseen voidaan joutua laatimaan testiajureita (*test drivers*) ja tynkämoduuleita (*test stubs*). Testiajurit mahdollistavat moduulien toteuttamien palveluiden kutsumisen ja tulosten tarkastelun. Tynkämoduulit esittävät aliohjelmaa, jota testattava moduuli kutsuu. Moduulitestaus on luonteeltaan lasilaatikkotestausta, eli moduulien rakenne tunnetaan yksityiskohtaisesti. [Rauhala, 2010].

Integrointitestauksessa on tarkoituksena testata moduulien välisten rajapintojen toimintaa. Integrointitestaus käyttää hyväkseen moduulitestauksen tuloksia. Integrointitestaus voi edetä rinnan moduulitestauksen kanssa. On huomioitava, että muuttujia on voitu määritellä globaalisti tiedostojen tai tietokantojen kautta [Rauhala, 2010].

Regressiotestauksessa todennetaan ohjelmistoon tehdyn korjauksen toimivuus ja varmennetaan, että uusia vikoja ei ole syntynyt muuttumattomiin ohjelmiston osiin korjauksen myötä [Rauhala, 2010].

Konfiguraatiotestauksia suoritetaan ohjelmistoille, jotka on suunniteltu toimimaan erilaisissa laite- ja käyttöjärjestelmäympäristöissä [Rauhala, 2010].

Testausta varten on oltava käsitys ohjelman syötteistä ja oikeasta lopputuloksesta. Tätä varten tarvitaan spesifikaatio, jonka perusteella testitapaukset voidaan suunnitella ja voidaan päätellä oikeat lopputulokset. Testaustulos riippuu syötteen lisäksi järjestelmän sisäisestä tilasta. Sisäinen tila tarkoittaa mm. ohjelman muuttujien ja rekisterien arvoja sekä levylle talletettuja tietoja. Testin onnistuminen edellyttää, että tulos on oikea ja että sisäinen tila on muuttunut oikein [Rauhala, 2010].

Testikattavuutta tai toisinsanottuna haluttua testipeittoa voidaan luokitella. Lausekattavuudella (*statement coverage*) tarkoitetaan, että ohjelman jokainen lause suoritetaan vähintään kerran. Todellisuudessa päästään harvoin tilanteeseen, jossa saavutetaan 100 % lausekattavuus. Polkujen kattavuudella (*path coverage*) kaikki ohjelman polkujen vaihtoehdot tulisi pystyä suorittamaan valituilla testitapauksilla. Polulla tarkoitetaan ohjelman kohtaa, jossa on kaksi tai useampia vaihtoehtoisesti suoritettavia lauseita (if-else, switch-case jne.). Täydellinen polkujen kattavuus on mahdollista saavuttaa vain yhden yksinkertaisen funktion sisällä. Ehtokattavuudessa (*condition coverage*) edellytetään, että päätöksen kaikkien osaehtojen on saatava molemmat arvonsa (tosi/epätosi). Päätöskattavuudesssa (*decision coverage*) edellytetään, että ehtorakenteen päätös saa vähintään kerran molemmat arvonsa (tosi/epätosi) [Rauhala, 2010].

4.2. JUnit

JUnit on avoimen lähdekoodin sovelluskehys, jolla voidaan ajaa toistettavia testejä. Jos ei käytä testisovelluskehystä voi testit kirjoittaa vaikka varsinaiseen lähdekoodiin kuten taulukon 23 ohjelmassa. Ominaisuuksiin kuuluu väitteiden (*assertion*) tekeminen testin odotettujen tuloksien tarkastamista varten. Testipetien (*test fixture*) tekeminen testien uudelleenkäyttöä varten ja testiajureita testien ajamista varten. JUnitin käyttö vaatii selityksien eli annotaatioiden (*annotation*) lisäämistä lähdekoodiin. Esimerkiksi testifunktio määritellään Test-annotaatiolla (`@Test`), jonka JUnitin testiajuri osaa tulkita ajettavaksi testiksi [Välimäki, 2011].

Javan annotaatiot ovat lähdekoodiin liitettävää metadataa, joita esimerkiksi kääntäjä voi käyttää tekemään tarkastuksia koodin oikeellisuudesta. Väitteitä taas tehdään käyttämällä `Assert`-luokan funktioita. `SetUp`-funktio muodostaa testipedin, joka on annotoitu Before-annotaatiolla (`@Before`), jolloin funktio ajetaan ennen jokaista testiä [Välimäki, 2011].

Esimerkki testien sisällöstä ja testipedistä löytyy taulukosta 10. Luokassa oleva `setup`-funktio alustaa testipedin eli testidatan. Tämä testidata on aina alustettu, kun JUnit ajaa jotakin testifunktiota. Samalla on myös mahdollista asettaa testattava ohjelma haluttuun tilaan, kuten esimerkiksi avaamaan tiedosto ja pitämään tiedostokahva varattuna. Myös odotetut tulokset olisi voitu laittaa testidataan. Testiluokat kannatta sijoittaa omiin luokkiinsa ja lähdekoodirakenteessa omaan hakemistoonsa erikseen varsinaisesta lähdekoodista.

```
1  public class MoneyTest {
2      private Money f12CHF;
3      private Money f14CHF;
4      private Money f28USD;
5
6      @Before public void setUp() {
7          f12CHF=new Money(12, "CHF");
8          f14CHF=new Money(14, "CHF");
9          f28USD=new Money(28, "USD");
10     }
11 @Test public void test1() {
12         Money result=f12CHF.add(f14CHF);
13         Money expected=new Money(26, "CHF")
14         assertTrue(expected.equals(result));
15     }
16 @Test public void test2() {
17         Money result=f28USD.add(f28USD);
18         Money expected=new Money(56, "USD")
19         assertTrue(expected.equals(result));
20     }
21 }
```

Taulukko 10. Esimerkki testipedin ja testausfunktioiden kirjoittamisesta lähdekoodiin. Esimerkki on muunnelma JUnitin dokumentaatiosta.

Kun testiä halutaan ajaa käytetään komentorivillä esimerkiksi seuraavanlaista komentoa `java.org.junit.runner.JUnitCore TestClass1`. Testit voidaan ajaa myös javalla tehdystä ohjelmasta, jolloin ohjelmasta kutsutaan JUnitia esimerkiksi `org.junit.runner.JUnitCore.runClasses(TestClass1);`. JUnit sisältää graafisessa käyttöliittymässä toimivan testiajurin `driver-junit.swingui.TestRunner`. JUnitista kannattaa lukea lisää http://junit.org/junit4/.

4.3. Testitapauksien generoinnista

Testitapausten tunnistamisen ja suunnittelun automatisoimista on syytä harkita tarkkaan, sillä ne suoritetaan yleensä vain kerran. On silti olemassa lukuisia testaustyökaluja testitapausten suunnittelun automatisoimiseksi. Ongelmaksi työkalua käytettäessä voi muodostua suuri testien lukumäärä, eikä työkalu osaa määritellä, mitkä testit ovat kaikista tärkeimpiä ajan säästämiseksi. Testitapausten generointivälineet ovat tosin tarkempia ja täsmällisempiä kuin ihminen [Vehkaperä, 2010].

Testitapausten generointivälineet voivat pohjautua koodiin, rajapintoihin tai määrittelyihin. Koodiin pohjautuva työkalu generoi testisyötteet tutkimalla ohjelmistokoodin rakennetta. Koodi muodostaa polun, joka koostuu lohkoista, joita määrittävät ehdot. Työkalu pystyy määrittämään jokaisen polkulohkon vaatimat loogiset ehdot automaattisesti ja näin saadaan samalla tietoa myös kattavuuden mittaamiseen [Vehkaperä, 2010].

Koodiin pohjautuvan generointityökalun käytössä täytyy huomioida se, ettei se sisällä testioraakkelia eli lähdettä oikeista odotetuista testien tuloksista. Näin ollen ei voida vertailla varsinaisen testin tavoin, olivatko saadut tulokset oikeita. Lisäksi tässä menetelmässä ongelmana voivat olla puuttuvat koodit, joita ei pystytä havaitsemaan [Vehkaperä, 2010].

Vaikka päädyttäisiin automatisoimaan suunnittelu, on syytä muistaa, että ihmistä tarvitaan edelleen priorisoimaan testit ja suunnittelemaan ne testit, joita työkalut eivät pysty laatimaan. Lisäksi kaikki edellä mainitut menetelmät voivat generoida liikaa testejä ja vaikuttavat näin ajankäyttöön ja ylläpidettävyyteen. Parhaimmassa tapauksessa hyödyt voivat silti olla vaivan arvoisia [Vehkaperä, 2010].

Suunnittelun automatisoimisen lisäksi voidaan automatisoida testien rakentamista, suorittamista tai testien lopputulosten vertailua odotettuihin tuloksiin. Suorituksen automatisoimiseksi ei välttämättä tarvita työkalua. Etenkin ohjelmat, jotka eivät ole suoraan käyttäjän kanssa tekemisissä, ovat helpompia automatisoida. Yksittäinen komentotiedosto voidaan ohjelmoida käynnistämään ohjelma ja suorittamaan testitulosten vertailun [Vehkaperä, 2010].

Vertailua voidaan suorittaa testaamisen aikana eli suorittaa dynaaminen vertailu tai tehdä vertailu suorituksen jälkeen sekä käyttää molempia vertailutapoja yhdessä. Dynaamisessa vertailussa ohjeet vertailuun ovat testiskriptissä. Ohjeet kertovat, mitä ja milloin verrataan sekä mihin verrataan. Vertailu voidaan yleensä pysäyttää kesken suorituksen tarkempaa tarkastelua varten ja lisätä saatu tulos odotetuksi lopputulokseksi. Testiskriptejä voidaan myös muokata manuaalisesti, mikä luonnollisesti edellyttää ohjelmointitaitoja, mutta siten voidaan lisätä testitapauksiin älykkyyttä. Voidaan esimerkiksi ohjelmoida vertailun lopettaminen tietynlaisen virheen esiintyessä, mikäli jatkaminen olisi turhaa [Vehkaperä, 2010].

4.4. Automaattiset testit

Automaattiset testit suorittavat täsmälleen samat operaatiot joka ajokerralla. Tämä poistaa inhimilliset virheet testejä ajettaessa. Automaatio myös säästää työaikaa, koska testien ajamiseen ei tarvita ihmistä. Tällöin testejä voidaan myös ajaa useammin, jolloin uudet ohjelmistovirheet huomataan nopeammin [Välimäki, 2011].

Toistettavuudesta saadaan se hyöty, että testejä voidaan ajaa peräjälkeen, jolloin testien tuloksien pitäisi olla joka ajokerralla samat. Jos tulokset vaihtelevat, on virhe joko ohjelmistossa tai testausautomaatiojärjestelmässä [Välimäki, 2011].

Uudelleenkäytettävyydellä tarkoitetaan, että testejä voidaan ajaa uudelleen ohjelmiston eri versioilla. Tällöin uusien versioiden ohjelmistovirheet huomataan mahdollisimman nopeasti. Testejä on joskus muutettava eri ohjelmiston versioiden välillä, mutta oletusarvoisesti testit pitäisi olla suunniteltu niin, ettei jokainen pieni muutos aiheuta usean testin muokkausta [Välimäki, 2011].

Testien automaattinen ajaminen myös nopeuttaa testien suoritusta, koska ihminen ei ole hidastamassa testien vaatimien toimintojen suoritusta. Testien suuresta määrästä johtuen nopeus on ensiarvoinen ominaisuus. Automaattisella testausjärjestelmälläkin ajettuna kaikkien testien ajaminen voi kestää tunteja [Välimäki, 2011].

4.5. Ohjelmien oikeellisuuden todistamisesta formaalilla verifioinnilla

Formaalit tarkistamiset (*formal verification*) ovat tapoja tarkistaa spesifikaatio tai suunnitelma (*design*) matemaattisesti. Formaalia tarkistamista voidaan tehdä teoreematodistus-menetelmillä (*theorem-proving methods*) ja mallin tarkastuksella (*model checking*) [Kung and Zhu, 2008].

Teoreematodistusmenetelmässä järjestelmän spesifikaatio koostuu joukosta esitteleviä lausumia (*declarative statements*) tai lauseita (*declarative sentences*). Nämä lausumat spesifioivat todellisen maailman ja/tai järjestelmän yksiköiden (*entities*) ja/tai olioiden (*objects*) ominaisuuksia (*properties*), niiden käyttäytymistä ja niiden riippuvuussuhteita (*relationships*). Matemaattisilla termeillä lausumien joukkoa kutsutaan teoriaksi (*theory*) ja se oletetaan todeksi koska lausumat totevat, mitä asiat ovat järjestelmässä. Ohjelmistotuotannossa lausumia kutsutaan ei-loogisina/epäloogisina (*nonlogical*) aksioomina, koska ne eivät ole loogisesti tosia, mutta oletetaan tosiksi todellisen sovelluksen (*real world application*) lakien mukaan. Esimerkiksi lauseita “jokaisella asiakkaalla on tili” ja “jokainen tili on jonkun asiakkaan omistama” ei voi todistaa logiisesti tosiksi, mutta ne voivat olla tosia jollekin pankkisovellukselle. Formaali tarkistaminen todistusteoreettisessa lähestymistavassa on todistamista että halutut järjestelmän ominaisuudet tai rajoitteet ovat loogisia seurauksia epäloogisista aksioomeista. Eli epäloogisista aksioomeista voidaan loogisesti johtaa halutut ominaisuudet tai rajoitteet [Kung and Zhu, 2008].

Mallin tarkastusta voidaan kutsua myös malliteoreettiseksi lähestymistavaksi. Mallin tarkastamisessa järjestelmä esitetään toiminnallisella mallilla (*operational model*), joka tyypillisesti kuvaa järjestelmän käyttäytymistä. Yleisesti käytetty toiminnallinen malli mallin tarkastamisessa on tilakone, joka koostuu järjestelmän tilaa kuvaavista kärjistä/solmuista (*vertices*) ja suunnatuista kaarista, jotka kuvaavat järjestelmän toimintoja, jotka aiheuttavat tilojen siirtymisiä. Jokainen järjestelmän tila on määritelty loogisilla tai ehdollisilla (*conditional*) lauseilla. Siis järjestelmä voi olla kyseisessä tilassa vain jos ehdot ovat tosia käyttäen järjestelmän attribuutteja. Formaali tarkastaminen mallin tarkastuksella alkaa aloittavalla järjestelmän tilalla ja luo tiloja operaatioita, eli siis kaaria käyttämällä. Halutut ominaisuudet tai rajoitteet tarkistetaan jokaista luotua tilaa kohden ja rikkeet (*violations*) raportoidaan [Kung and Zhu, 2008].

Nykyisin mallin tarkastusta ollaan käytetty ennemminkin koodin tai toteutuksen tarkistamiseen kuin spesifikaation tarkistamiseen. Tälle ollaan annettu termi ohjelmiston mallin tarkistus (*software model checking*). Tällöin malli rakennetaan koodista eikä spesifikaatiosta. Tällaisen mallin valmistaminen voi olla täysin manuaalista tai puoliautomaattista (*semi-automatic*) [Kung and Zhu, 2008].

Digitaalijärjestelmiä suunniteltaessa laaditaan järjestelmästä järjestelmäkuvaus, joka koostuu järjestelmän toiminnallisuuden yksityiskohtaisesta määrittelystä. Verifioinnin tavoitteena on tarkastaa, toteuttaako annettu järjestelmäkuvaus tietyn ominaisuuden. Esimerkiksi "järjestelmä ei voi päätyä lukkiumatilaan" eli järjestelmää ei ole mahdollista ajaa millään syötejonolla tilaan, jossa se ei kykene tekemään mitään. Ongelma voidaan periaatteessa ratkaista simuloimalla järjestelmän kaikkia mahdollisia suorituksia. Nykyisissä järjestelmissä mahdollisten suoritusten määrä on kuitenkin huikea, mistä johtuen ongelmaksi voi muodostua tila-avaruusräjähdys [Järvisalo, 2004].

Mallintarkastus on kirjoittajasta riippuen synonyymi sanalle verifiointi, tai verifioinnin alalaji, jossa verifioitava ominaisuus ilmaistaan loogisella kaavalla. Symbolisessa mallintarkastuksessa tila-avaruusräjähdys pyritään välttämään luopumalla järjestelmän tilojen eksplisiittisestä esittämisestä. Tutkinnan kohteena on ollut toteutuvuustarkastukseen perustuva symbolinen mallintarkastus. Rajoitettu mallintarkastus BMC (*Bounded Model Checking*) on toteutuvuustarkastusmenetelmiin pohjautuva symbolinen mallintarkastusmenetelmä [Järvisalo, 2004].

Tavoitteena BMC:ssä on tarkastaa, toteuttavatko järjestelmän kaikki enintään k:n mittaiset suoritukset tietyn ominaisuuden. Ideana on kuvata halutun ominaisuuden negaatio sekä järjestelmän k:n askeleen tilasiirtymät lauselogiikan lauseena ja tarkastaa, onko tälle lauseelle mallia toteutuvuustarkastinta käyttäen. Iteratiivisesti k:n arvoa kasvattaen yritetään löytää vastaesimerkki, joka osoittaa järjestelmän kuvauksen virheellisyyden. Menetelmä soveltuu erityisesti järjestelmien virheiden etsintään. Järvisalo [2004] tarjoaa selventävän esimerkin artikkelissaan ja en toista sitä tässä.

5. Virheiden korjaamiseen kehitetyt menetelmät

Tässä luvussa tarkastellaan virheellisen koodin korjauksessa käytettäviä menetelmiä. Kohdassa 5.1 käsitellään geneettisen algoritmin toteutusta GenProg. Kohdassa 5.2 semanttisen analyysin toteutusta SemFix. Lisäksi kohdassa 5.3 tarkastellaan uudempaa semanttiseen analysiin perustuvaa menetelmää, jota käytetään DirectFix ohjelmassa.

Kaikki menetelmät riippuvat virheen oikeasta paikantamisesta ja riittävästä ja oikeanlaisesta testipeitosta. Mikään menetelmä ei toimi sellaisiin virheisiin, jotka eivät ole toistettavissa tai esiintyvät sattumanvaraisesti. Myös ongelmat, jotka johtuvat ohjelmien suunnitteluvirheistä ovat automaattisen virheenkorjauksen korjausmahdollisuuksien ulkopuolella tällä hetkellä. Näihin ongelmiin yritetään kuitenkin kehitellä ainakin osittain automaattisesti toimivia ratkaisuja, kuten voi tarkistaa AutoFix:in kehittäjien artikkelista. Myös eri menetelmiin käytetty testiaineisto vaihtelee, ja eri menetelmien tehokkuusluokitus vaihtelee valitun testiaineiston mukaan. Myös korjausten ylläpidettävyys ja vertailu käsillä tapahtuvaan korjaukseen on tutkinnan kohteena. [Pei et al., 2014].

5.1. Geneettinen algoritmi

Geneettinen ohjelmointi on saanut inspiraationsa luonnossa tapahtuvasta evoluutiosta. Geneettisellä ohjelmoinnilla luodaan muunneltuja populaatioita alkuperäisestä ohjelmasta. Populaation yksilöiden hyvyys halutun tehtävän suorittamiseen arvioidaan tehtäväkohtaisella hyvyysfunktiolla. Korkeimman hyvyysarvon saaneet tehtävän läpäisseet yksilöt otetaan jatkokehittelyyn. Biologisten mutaatioiden ja risteytysten laskennalliset vastaavuudet tuottavat korkean hyvyysarvon ohjelmamuunnelmia. Kehitystä iteroidaan kunnes tarpeeksi hyvänlaatuinen ohjelma on löydetty [Weimer et al., 2010].

Virheenkorjauksessa luodaan automaattisesti muunneltuja populaatioita alkuperäisestä ohjelmasta, joka ei läpäissyt haluttua testitapausta, eli testitapaus on läpäisyä vaativa tehtävä. Parhaita tuloksia saavuttava geneettisiä algoritmejä käyttävä toteutus on GenProg [Weimer et al., 2010].

Evoluutionallisten algoritmien, kuten geneettinen ohjelmointi, ongelmana on suuri kokeiltavien näytteiden hakuavaruus oikein toimivan ohjelman löytymiseen. GenProgin kehittäjät kiersivät tämän ongelman olettamalla, että ohjelman korjaava rakenne löytyy jo korjattavasta ohjelmasta, ja siten rajoittivat mutaatioiden käyttämään uudelleen rakenteita, jotka sijaitsevat muissa kohdissa korjattavaa ohjelmaa. Lisäksi mutaatiot on rajoitettu vain kohtiin, jotka ovat olennaisia epäonnistuneelle testille, eli virheelle [Weimer et al., 2010].

GenProgin käyttämä tietorakenne ohjelman rakenteesta on abstrakti syntaksipuu AST (*abstract syntax tree* [Baxter et al., 1998]). AST tunnetaan myös abstraktina jäsennyspuuna, ja sitä on käytetty muun muassa kääntäjien välikielen tietorakenteena. AST on valittu, koska se voi esittää häviöttömästi rakenteellisia ohjelmia ja operaatiot puurakenteessa ovat hyvin tutkittuja geneettisessä ohjelmoinnissa. Laajennettavuuden eli skaalautuvuuden vuoksi GenProgissa koodilausumaa

(*statement*) käsitellään perusyksikkönä tai geeninä. Nämä geenit ovat AST-puun solmuja ja lehtiä [Weimer et al., 2010].

GenProg tuottaa variaatioita ohjelmasta mutaatio-operaatioilla, kuten poisto, vaihto ja lisäys. Lisäksi on eroavaisuusoperaatio, joka vaihtaa satunnaisesti valitun alipuun kokonaan [Weimer et al., 2010].

GenProg ei tee mutoksia tietotyyppien esittelyihin. Tästä seuraa rajoite sellaisille korjauksille, jossa paras korjaus olisi tietorakenteen muuttaminen. GenProg voi kuitenkin vaihtaa muuttujaa käyttävän koodilausuman paikkaa muuttujan määrittelyalueen ulkopuolelle ja siten luoda muunnelman, joka ei käänny [Weimer et al., 2010].

GenProgin sopivuusfunktio kääntää muutetun ohjelman ja suorittaa testitapaukset. Se palauttaa painotetun summan hyväksytyistä testitapauksista. Painotus on sellainen, että aiemmin epäonnistuneet, mutta muunnoksella toimivat testitapaukset ovat painoltaan vähintään samanarvoiset aiemmin onnistuneiden testitapausten kanssa. Ohjelmat, jotka eivät käänny, saavat painotuksekseen nollan [Weimer et al., 2010].

Geneetinen ohjelmointi saattaa lisätä merkityksettömiä muutoksia, jonka vuoksi GenProg suorittaa ohjelman analysointia muutosten poistamiseen itse korjauksesta. Tähän GenProg käyttää puurakenteisia erottelualgoritmeja ja delta-korjaus tekniikoita. Täten jälkiprosessointivaiheessa saadaan aikaiseksi yksinkertaistettu ohjelmapaikkaus, joka alkuperäiseen ohjelmaan sovellettuna saa korjattavan ohjelman suorittamaan testitapaukset onnistuneesti [Weimer et al., 2010].

Puurakenteisella erottelulla GenProg kerää muutokset joukoksi. Delta-korjauksessa testitapaus ajetaan uudestaan vain puolella alkuperäisistä muutoksista. Mikäli testi tulee suoritetuksi, ajetaan testi taas uudestaan, ja edellisen kerran muutoksista otetaan puolet pois. Operaatiota toistetaan, kunnes testi epäonnistuu. Testi suoritetaan uudestaan, mutta edelliseen testiin verrattuna muutoksista ei oteta puolia pois, vaan ¼. Testi suoritetaan uudestaan, kunnes testi epäonnistuu. Taas pienennetään poistettavien muutosten määrää, eli otetaan muutoksista 1/8 pois. Operaatiota toistetaan, kunnes vain yhden muunnoksen poistaminen aiheuttaa testin epäonnistumisen. Delta-korjaus löytää minimaalisen muutosten joukon ajassa $O(n^2)$. Kokeilujen mukaan lopullinen korjaus on tyypillisesti suuruusluokan verran pienempi kuin alkuperäinen korjaus [Weimer et al., 2010].

Esimerkkinä, minkälaisia muutoksia koodissa tapahtuu, käytän GenProgin kehittäjien antamaa Microsoftin Zune-ohjelman virheen korjausta. Alkuperäinen virheellinen koodi on esitetty taulukossa 11, jossa ohjelma jää jumiin, jos syötteenä annettu `days` on karkausvuoden viimeinen päivä [Weimer et al., 2010].

Virheellisessä Zune-koodissa positiiviset testitapauksen testit suorittavat rivit 1-8 ja 11-18. Negatiivisessa testitapauksessa suoritetaan rivit 1-16 ja sen jälkeen jäädään toistamaan rivejä 3,4,8 ja 11 loputtomasti. Negatiivisissa testitapauksissa syötteinä ovat 366 ja 10593 [Weimer et al., 2010].

1	`void zunebug(int days) {`
2	`    int year = 1980;`
3	`    while (days > 365) {`
4	`        if (isLeapYear (year) ){`
5	`            if (days > 366) {`
6	`                days -= 366;`
7	`                year += 1;`
8	`                }`
9	`            else {`
10	`            }`
11	`        }`
12	`        else {`
13	`            days -= 365;`
14	`            year += 1;`
15	`        }`
16	`    }`
17	`    printf(“the year is %d\n”, year);`
18	`}`

Taulukko 11. Virheellisesti toimiva alkuperäinen Zune ohjelma.

Korjauksen etenemisessä tarkastellaan muunnelmaa (*variant*) V, joka on alussa identtinen alkuperäiseen ohjelmaan. Ensimmäisessä sukupolvessa (*generation*) kaksi operaatiota muuttavat muunnelmaa ja tämä näkyy taulukossa 12 lisäyksinä alkuperäisen koodin rivien 6 ja 7 sekä 1+ ja 11 väliin. Ensimmäinen lisäys on alipuun lisääminen. Toinen operaatio on rivin lisäys. Tällä 1. sukupolven muunnoksella negatiivinen testi arvolla 366 suoritetaan hyväksytysti. Muutokset kuitenkin aiheuttivat sen että aiemmin suoritettuja testejä ei kaikkia saatu hyväksytysti suoritettua [Weimer et al., 2010].

```
5             if (days > 366) {
6                 days -= 366;
7   if (days > 366) { // insert #1 // orig    year += 1;
8       days -= 366; // insert #1 // orig    }
9       year += 1;   // insert #1 // orig    else {
10                } // insert #1 // orig    }
11        }
12        else {
13            days -= 365;
14            year += 1;
15    days -= 366; // insert #2 // orig    }
16    }
17    printf("the year is %d\n", year);
18 }
```

Taulukko 12. Ensimmäisen sukupolven variantti.

Muunnelma V säilyy muuttumattomana sukupolvet 2-5, mutta kuudennessa sitä muutetaan. Muutokset näkyvät taulukossa 13. Tässä vaiheessa kaikki testit suoritetaan hyväksytysti ja korjauksen haku lopetetaan ja muunnelmaa V pidetään pääasiallisena korjauksena [Weimer et al., 2010].

```
5   if (days > 366) {
6   // days -= 366; // delete
7   // if (days > 366) { // delete
8   // days -= 366; // delete
9   // year += 1; // delete
10  // } // delete
11  year += 1;
12  }
13  else {
14  days -= 366; // insert
15  }
16  days -= 366;
```

Taulukko 13. Kuudennen sukupolven muunnelman V muutokset.

Seuraavassa vaiheessa korjausta yritetään minimoida. Lopullinen korjaus on taulukossa 14. Lopullinen korjaus saavutetaan yhdellä poistolla ja yhdellä lisäyksellä [Weimer et al., 2010]. Lisätty ja poistettu rivi ovat kuitenkin täysin samanlaiset, joten korjauksessa on pikemminkin kyse yhdestä

siirto-operaatiosta. Muutoksessa kyseessä on kuitenkin muuttuvien rivien määrä Unixin `diff`-työkalulla ilmoitettuna.

1	`void zunebug_repair(int days) {`
2	`    int year = 1980;`
3	`    while (days > 365) {`
4	`        if (isLeapYear (year) ){`
5	`            if (days > 366) {`
6	`                // days -= 366;    // deleted`
7	`                year += 1;`
8	`                }`
9	`            else {`
10	`            }`
11	`            days -= 366;    // inserted`
12	`        } else {`
13	`            days -= 365;`
14	`            year += 1;`
15	`        }`
16	`    }`
17	`    printf(“the year is %d\n”, year);`
18	`}`

Taulukko 14. Lopullinen GenProg-ohjelman tekemä korjaus Zune-ohjelmaan.

Esimerkkinä käytetyn Zune-katkelman kaltaisten kohtien korjaus kestää noin 42 sekuntia, sisältäen korjauksen, korjauksen minimoimisen sekä kääntämiset. Lisäksi käännetyt variantit testataan ja käytetyt testitapaukset, joista 2 hylättyä ja 5 hyväksyttyä kuuluvat mainittuun 42 sekunnin korjauksen luontiin. Taulukosta 15 voi tarkastella koodirivien, painotetun polun, korjausajan ja korjauksen koon riippuvuuksia. Kyseisen taulukon ohjelma `look utx 4.3` vaati 11 rivin muutoksen, ja on pisin korjaus [Weimer et al., 2010].

Ohjelma	Koodi-rivejä	Paino-tettu polku	Ohjelman kuvaus	Virhe	Aika	Sopi-vuus-arviot	Korjauk-sen koko
gcd	22	1.3	Euclid's algorithm	Infinite loop	153	45.0	2
zune	28	2.9	MS Zune excerpt	Infinite loop	42	203.5	4
uniq utx 4.3	1146	81.5	Duplicate filtering	Segmentation fault	34	15.5	4
look utx 4.3	1169	213.0	Dictionary lookup	Segmentation fault	45	20.1	11
look svr 4.0	1363	32.4	Dictionary lookup	Infinite loop	55	13.5	3
units svr 4.0	1504	2159.7	Metric conversion	Segmentation fault	109	61.7	4
deroff utx 4.3	2236	251.4	Document processing	Segmentation fault	131	28.6	3
nullhttpd 0.5.0	5575	768.5	Webserver	Heap buffer overrun	578	95.1	5
indent 1.9.1	9906	1435.9	Source codeformatting	Infinite loop	546	108.6	2
flex 2.5.4a	18775	3836.6	Lexical analyzer generator	Segmentation fault	230	39.4	3
atris 1.0.6	21553	34.0	Graphical tetris game	Stack buffer overrun	80	20.2	3

Taulukko 15. Vertailua GenProgin korjaamista ohjelmista ja korjauksista. Käytetyissä testitapauksissa oli 5-6 positiivista ja 1-2 negatiivista testiä. Sopivuusarviot-sarakkeessa ilmoitetaan kuinka monta kertaa sopivuus funktiota on kutsuttu ennen kuin korjaus on löytynyt, mutta se on keskiarvoistettu vain onnistuneisiin yritelmiin. Korjauksen koko sarakkeessa on `diff`-ohjelman ilmoittama muuttuvien rivien lukumäärä viallisen ja korjatun ohjelman välillä.

Yritteissä populaation koko on 40 ja sukupolvien määrä on rajoitettu 20:een. Ensimmäisten 10 sukupolven aikana globaali mutaatioiden suhde on 0.06. Jos pääasiallista korjausta ei tähän mennessä ole löydetty, nykyinen populaatio hävitetään. Loppujen 10 sukupolven luonnissa globaali mutaatioiden suhde puolitetaan 0.03:en, ja mikäli mahdollista, painotettu polku rajoitetaan vain niihin lausumiin, jotka käydään lävitse negatiivisissa testitapauksissa. Yritteen haku pysähtyy, jos se löytää pääasiallisen korjauksen. Jokaiselle ohjelmalle suoritettiin 100 yritettä, muistaen (*memoizing*) sopivuudet siten, että yhden yritteen sisällä kahta yksilöä erilaisilla AST:llä mutta samalla

lähdekoodilla ei arvioida kahdesti. Samoin yksilöitä, jotka kopioidaan muuttumattomina seuraavaan sukupolveen, ei arvioida uudestaan [Weimer et al., 2010].

Pääasiallisen korjauksen löytymisen jälkeen kyseisen korjauksen minimointi on suhteellisen nopeaa ja on vaatinut keskiarvoisesti alle 5 sekuntia aikaa. Kaikki yritelmät eivät kuitenkaan johda onnistuneeseen korjaukseen. Taulukon 15 ohjelmien korjauksien yritelmistä 60 prosenttia tuotti pääasiallisen korjauksen. Yritelmät ja niiden muunnelmien testaaminen ja sopivuuksien arvioinnit voidaan kuitenkin tehdä toisistaan riippumattomasti. Täten menetelmän tehokkuus kasvaa, kun käytettävien prosessoriytimien määrää kasvatetaan [Weimer et al., 2010].

Yli puolet korjauksen luomiseen vaaditusta ajasta kulutetaan sopivuuden arvioinnissa, kun käännettyjä muunnelmia ajetaan testitapausten kanssa. Ohjelmilla, joilla testisarjat ovat laajoja, voi arvioinnin kesto olla huomattava. Siten vaikuttavin tekijä tämän menetelmän skaalautuvuudessa on sellaisten sopivuusarvioiden määrä jotka pitää tehdä korjauksen löytämiseksi. Vaadittujen sopivuusarvioiden määrä riippuu hakuavaruuden koosta ja hakustrategian tehokkuudesta. Jokainen sopivuusarviointi edustaa yksittäistä koetinta (*probe*). GenProgin kehittäjät olettavat että painotetun polun koko on hyvä hakuavaruuden koon esittäjä. Yksi peruste tälle on vain niiden lausekkeiden muokkaaminen, jotka ovat painotetulla polulla [Weimer et al., 2010].

Rajoituksena GenProgin menetelmälle on, että virheen on oltava toistettavissa ja ohjelma käyttäytyy ennakoitavasti. Näitä rajoituksia voidaan lieventää ajamalla testitapauksia useita kertoja, mutta jos ohjelma käyttäytyy satunnaisesti, tulee sen korjaaminen hankalaksi. GenProg voi myös rikkoa tärkeää toiminnallisuutta korjauksissaan, jos testitapaukset eivät testaa kyseisiä toimintoja. Useimmiten kuitenkin testejä on yleensä vaadittua enemmän ja ne hidastavat sopivuuden arviointia ja haittaavat korjauksen hakua. GenProgin käytössä myös oletetaan, että hyväksytyt testit menevät eri reittiä koodissa kuin epäonnistuneet testit. Jos kyseiset reitit ovat täysin päällekkäisiä, ei painotettu polku pysty ohjaamaan GenProgin muutoksia niin tehokkaasti kuin eriävien reittien tapauksessa. Lisäksi oletetaan, että korjaavat lausekkeet ovat jo mukana korjattavassa ohjelmassa. Havaintona on myös ollut, että virheen paikka on harvoin sama kuin mistä lisättävä eli korjaava koodi löytyy. Testattaessa menetelmää yli puolet korjauksista sisälsi joko koodin lisäystä tai vaihtoa. Siten oikeanlaisten korjausten paikantaminen on kriittisen tärkeää [Weimer et al., 2010].

GenProgin sopivuusfunktio ei välttämättä ole tarpeeksi tarkka evolutiivisen haun ohjaamiseen monimutkaisemmissa ongelmissa. Sopivuusfunktion toiminta aiheuttaa ongelmia esimerkiksi moniajollisissa kilpailutilanteissa, ja siten ohjaa korjauksia väärään suuntaan. Korjausten laatu vastaa hyvin pitkälle ihmisten tekemiä korjauksia, mutta korjaukset pitäisi myöskin dokumentoida tai mahdollisesti todistaa korjausten ominaisuuksia. Käytettyä AST esitystapaa voi myös parantaa esimerkiksi sisällyttämään tietorakenteiden määrittelyt ja muuttujien esittelyt [Weimer et al., 2010].

GenProg ei suoraan muuta ilmaisuja kuten `1-2` tai `!P`, eikä se muuta matalan tason kontrollin ohjauksia kuten `break`, `goto` ja `continue`. Näistä rajoituksista sallittuihin ohjelman muutoksiin seuraa, että GenProg ei ikinä luo syntaktisesti väärin muodostettuja ohjelmia kuten esimerkiksi parittomia sulkeita. GenProg voi kuitenkin luoda ohjelmia, jotka eivät käänny semanttisen virheen

takia, kuten esimerkiksi siirtämällä muuttujan käytön muuttujan määrittelyalueen ulkopuolelle [Goues et al., 2012].

Painotettu polku on sarja pareja ⟨lausuma, paino⟩, joka rajoittaa mutaatio-operaatiot pienemmälle, enemmän painotetulle, ohjelmapuun alijoukolle. Lausumia, joiden paino on nolla, ei koskaan muuteta, mutta niitä voidaan kopioida painotetulle polulle mutaatio-operaatioilla siten, että niiden paino nousee. Jokaisella uudella muunnelmalla alkuperäisestä ohjelmasta on sama määrä lausumien ja painojen pareja ja sama painojen järjestys painotetussa polussaan kuin alkuperäisellä ohjelmalla. Tämä on välttämätöntä risteytys (*crossover*) operaation takia [Goues et al., 2012].

Rakennettaessa painotettua polkua jokaiselle lausumalle annetaan ainutlaatuinen tunniste, joka merkitään lokiin jokainen kerta, kun lausumaa suoritetaan. Monistetut lausumat poistetaan painotetusta polusta, koska GenProgin kehittäjät eivät usko, että lausumat, joita suoritetaan esimerkiksi silmukassa, ovat hyviä kohtia korjaukselle. Lausumien suoritusjärjestys kuitenkin kirjoitetaan ylös, jolloin painotettu polku on järjestetty jono, eikä vain joukko paino, lausuma pareja. Jokainen lausuma, joka on suoritettu epäonnistuneessa testitapauksessa on ehdokas korjaukselle, saaden aluksi maksimi painon. Niiden lausumien painoja, joita suoritetaan myös hyväksytyissä testitapauksissa tullaan laskemaan [Goues et al., 2012].

GenProgin kehittäjien mielestä painotetun polun käyttäminen on pakollista suurimpaan osaan heidän tutkimiensa ohjelmien virheiden korjaamiseen. Ilman sitä, hakuavaruus kasvaisi liian suureksi tehokkaaseen korjauksen hakemiseen. Tästä huolimatta GenProgin kehittäjät uskovat myös, että virheen paikantaminen on vaikea ja selvittämätön ongelma ja on olemassa tietyntyyppisiä virheitä, joita on vaikea tai mahdoton paikallistaa. Täten GenProg voi tulevaisuudessa olla tehokkaampi, kun se voi käyttää pitkälle kehittyneitä virheiden paikannus menetelmiä [Goues et al., 2012].

Valinta, jolla päätetään, mitkä muunnelmat kopioidaan, eli jatkavat seuraavalle sukupolvelle, tapahtuu aluksi muunnelman saaman sopivuus arvon perusteella. Kääntymättömät ja sellaiset muunnelmat, jotka eivät läpäise yhtään testiä saavat sopivuusarvokseen nollan ja hylätään. Lopuista jäljelle jääneistä puolelle GenProgille populaation koko annetusta parametrin arvosta tehdään uusi ”parittelualla” (*mating pool*) valintastrategian kautta. Valinnassa yksilön todennäköisyys jatkovalintaan on suoraan verrannollinen sen suhteelliseen sopivuuteen. Lisäksi on käytetty turnajaisvalintaa, jossa pieniä alijoukkoja populaatiosta on valittu satunnaisesti, ja kunkin joukon sopivin yksilö on valittu seuraavaan sukupolveen. Tätä prosessia iteroidaan kunnes uusi populaatio on saatu valituksi. Kaksi operaatiota, mutaatio- ja risteytys-operaatio luovat ”parittelu altaasta” uusia variantteja [Goues et al., 2012].

Muutatio-operaatiolla on pieni mahdollisuus muuttaa mitä tahansa lausumaa painotetulla polulla. Muutokset muuttavat myös kyseisen ohjelman AST:tä. Lausumia muutetaan niiden todennäköisyyksien yhtäsuuruudella niiden painon kanssa. Yksilön mutaatioiden maksimimäärä määrätään parametrina annetun globaalin mutaatio asteen arvoilla. Mutaatio-operaatioita poisto, vaihto ja lisäys arvotaan tasaisesti ilman painotuksia. Lisäyksen ja vaihdon kanssa arvotaan lisäksi jokin lausuma mistä tahansa kohdasta korjattavaa ohjelmaa, eikä vain painotetulta polulta. Poisto

mutaatio muuttaa poistettavan lausuman tyhjäksi lohkoksi ja voi siten muuttua myöhemmissä mutaatio-operaatioissa. Kaikissa tapauksissa uusi lausuma säilyttää vanhan lausuman painon jotta yhtenäisyys painojen ja polun pituuden kanssa eri muunnelmien kesken säilyisi, ja koska lisätyt tai vaihdetut lausumat voivat tulla painotetun polun ulkopuolelta, jolloin niillä ei ole omaa painoa [Goues et al., 2012].

Risteytyksessä yhdistetään osia kahdesta ”vanhemmat” muunnelmasta ja kehitetään näistä uusia jälkeläisiä. Jokainen selviytynyt muunnelma populaatiosta käy risteytysoperaatiossa. Muunnelma on kuitenkin vain kerran vanhempi populaation sukupolvea kohden. Risteytykseen valitaan vain niitä lausumia, jotka sijaitsevat painotetulla polulla. Painotetulta polulta valitaan katkaisupiste, jonka jälkeiset lausumat vaihdetaan [Goues et al., 2012]. Kahdesta vanhemmasta synnytetään risteytys operaatiolla kaksi jälkeläistä. Jokainen vanhempi ja jokainen jälkeläinen käyvät tämän jälkeen vielä lävitse mutaation yhden kerran ja siirtyvät tulevan sukupolven populaatioon [Forrest et al., 2013].

Sopivuusfunktio ohjaa valintaa muunnelmista, jotka etenevät seuraavalle sukupolvelle. Samalla se sisältää myös korjauksen haun lopettamiskriteerit onnistuneen haun tapauksessa. Sopivuusfunktio käyttää testejä arvioidessaan muunnelman soveltuvuutta. Jokainen alunperin onnistunut testitapaus saa painon annetusta parametrista W_{PosT}. Onnistuneet alunperin epäonnistuneet testit saavat painon W_{NegT}. Sopivuusfunktio on vain kyseisten testien onnistuneiden testien kertominen annetuilla painoilla ja tulot lasketaan lopuksi yhteen. Eli sopivuusfunktio kaavana esitettynä

$$\text{fitness}(P)=W_{PosT} * |\{t \in PosT | P \text{ passes } t\}| + W_{NegT} * |\{t \in NegT | p \text{ passes } t\}|,$$

jossa P on ohjelma, t on onnistunut testi, $PosT$ on alun perin positiivisesti suoritettujen testien joukko ja $NegT$ on alun perin epäonnistuneiden, mutta nykyisellä muunnelmalla onnistuneiden testien joukko. Koska testitapaukset vahvistavat korjauksen oikeellisuuden, testisarjan valintaa kannattaa harkita tarkkaan. Jos alunperin hyväksyttyjä testejä ei ole mukana yhtään, voi GenProg tehdä muutoksia, joilla kriittistä toiminnallisuutta ei enää ole. Lisäksi testisarjan valinta voi vaikuttaa 80 prosentin verran aikaan, jolla korjaus löytyy [Goues et al., 2012].

Korjauksen minimoinnissa käytetään C-kääntäjälle tehdyn välikielen CIL:n (*C-intermediate language* [Necula et al., 2002], ei ole tässä yhteydessä sama kuin *common intermediate language*) AST:en kanssa toimivaa muokattua DIFFX XML erottelualgoritmia. Täten saadaan minimointi suoriutumaan tehokkaammin verrattuna todelliseen syntaksiin, jota `diff`–työkalu käyttää. Muokattu DIFFX luo listan puurakenteisista editointi-operaatioista kuten ”siirrä alipuu solmusta X solmun Z lapseksi Y”. Tämänlainen enkoodaus on tyypillisesti lyhyempi kuin vastaava `diff` paikkaus. Lisäksi puupohjaiset editoinnit eivät koskaan aiheuta syntaktisesti huonosti muodostettuja ohjelmia. Kun minimointi on valmis, voidaan DIFFX editoinnit automaattisesti muuttaa `diff` paikkauksiksi [Goues et al., 2012].

Korjausten laatua ja hyväksyttävyyttä ei kuitenkaan pidä jättää pelkästään hyväksyttyjen testien varaan. GenProg voi luoda korjauksia jotka huonontavat ohjelman arkkitehtuuria (*design of the system*) tai tekevät siitä hankalamman ylläpitää. Näiden asioiden arvioiminen automaateilla on tällä hetkellä vielä vaikeaa. Joitakin korjauksia, joita saatetaan tehdä riittämättömillä testisarjoilla, ja jotka

siten vaarantavat toiminnallisuutta tai aiheuttavat haavoittuvuuksia, voidaan arvioida esimerkiksi ohjeellisilla kuormituksilla (*indicative workloads*) [Goues et al., 2012].

GenProgin kehittäjät havaitsivat myöhemmin, että jokaisen muunnelman esittäminen kokonaisina AST rakenteina yhdistettynä painotettuun suorituspolkuun rajoitti skaalautuvuutta. Siksi muunnelma kannattaa esittää korjauksena (*patch*), eli sarjana editointi-operaatioita alkuperäiseen ohjelmaan nähden. Populaatioista, joiden koot olivat 40-80 yksilön suuruisia, tehdyt AST:t eivät mahtuneet 1.7 gigatavun päämuistiin. GenProgin toiminnan testaamiseen käytetystä testidatasta eli vikoja sisältävistä ohjelmista, tehdyistä korjauksista puolet olivat 25 rivin tai alle käsittäviä korjauksia. Siten kaksi toisistaan riippumatonta muunnelmaa eroavat toisistaan enimmillään 2*25 rivin verran muiden AST solmujen ollessa samoja. Siten muunnelmat vievät vähemmän tilaa paikkaus-muodossa, koska muuttumattomasta koodista ei tehdä turhia kopioita [Forrest et al., 2013].

Lisää optimointia on käytetty muunnelmien testaamiseen, koska muunnelmien testauksessa GenProg kuluttaa eniten aikaa. Testausta on rinnakkaistettu, jos se on havaittu mahdolliseksi. Lisäksi aluksi testataan vain korjauksen kohteena olevalla testillä ja satunnaisesti valitulla supistetulla joukolla jo läpäistyjä testejä. Vain ne muunnelmat jotka täysin läpäisevät supistetun testijoukon testataan myöhemmin koko testisarjalla [Forrest et al., 2013].

Alkuperäiseen versioon verrattuna GenProg:sa on paranneltu myös mutaatio-operaatioita. Lisäys ja korvaus operaatiot saattoivat aiemmin lisätä muuttujia, jotka olivat määrittelyalueensa ulkopuolella. Tämänkaltaiset muutokset on nyt suljettu pois ja siten sellaisia muunnelmia, jotka eivät käänny, esiintyy vähemmän [Forrest et al., 2013].

5.2. Semanttinen analyysi

Semanttisella analyysillä yritetään syntetisoida koodia kohtaan, jossa virhe esiintyy. Toteutus SemFix käyttää symbolista suorittamista (*symbolic execution*), rajoitteiden selvitystä (*constraint solving*) ja ohjelman syntetisointia (*program synthesis*) automaattiseen virheiden korjaukseen. SemFixin lähestymistavassa vaatimus testijoukon suoritukselle on muotoiltu rajoitteeksi (*constraint*). Rajoite ratkaistaan iteroimalla korjausilmausten kerroksittaista avaruutta. Kerroksittaminen on tehty korjauskoodin monimutkaisuuden mukaan [Nguyen et al., 2013].

SemFix on rajoitteisiin perustuva semanttinen lähestyminen ohjelman korjaamiseksi. Kohdasta, josta ohjelmaa pitää korjata, johdetaan rajoitteita korjaavalle lauseelle (*expression*), jotta muutettu ohjelma suorittaa kaikki annetut testitapaukset. Korjausrajoitteet (*repair constraint*) generoidaan ohjatun symbolisen suorittamisen kautta ja korjattava lause saadaan ohjelman syntetisoinnilla. SemFixin kehittäjät uskovat symbolisen suorittamisen rajoittavan skaalautuvuutta käsiteltävien ohjelmien kokoon [Nguyen et al., 2013].

SemFix on yhdistelmä kolmesta olemassa olevasta tekniikasta. Virheen paikantamiseen käytetään virheen eristämistä etäisellä ohjelman näytteistyksellä, jossa tilastollisesti arvioidaan käytyjen lauseiden virheellisyyttä. Lauseet on listattu virheen todennäköisyyden mukaan, joista SemFix tarkastelee yhtä lauseketta palautetuista arvioiduista lausekkeista kerrallaan. Lauseketason

spesifikaatiopäättelyä (*Statement-level specification inference*) käytetään virheellisen lausekkeen oikean spesifikaation selville saamiseksi. Idea on sama kuin enkeli-debukkauksessa (*angelic debugging*, katso kohta 2.7), jossa muutetaan ilmaisu (*expression*) ei-deterministiseksi ilmaisuksi. Tämä mahdollistaa luomaan tulosteen, joilla testi suoritetaan hyväksytysti, kaikilla syötteillä virheelliselle lausekkeelle. Ohjelman syntetisointi (*Program synthesis*) tuottaa keinotekoisesti lausekkeen, joka toteuttaa aiemmilla tekniikoilla löydetyn spesifikaation [Nguyen et al., 2013].

Toisen ja kolmannen tekniikan vuorovaikutus on SemFixin pääasiallinen idea. Lauseketason spesifikaatio rajoittaa merkittävästi hakuavaruutta ja asettaa ongelman komponenttipohjaiselle ohjelman syntetisoinnille, joka perustuu rajoitteen ratkaisemiseen. SemFix käyttää myös kahta tärkeää suorituskykyoptimointia. Rajoitetta ei luoda koko testisarjalle, vaan supistetulle testijoukolle, johon lisätään sekaan testejä inkrementaalisesti. Toinen optimointistrategia on aloittaa niiden ilmaisujen joukon, jotka voidaan luoda synteettisesti, tutkiminen yksinkertaisimmasta ilmaisusta siirtyen kohti monimutkaisempia ilmaisuja virheen korjauksessa. Siten SemFix löytää yksinkertaiset korjaukset nopeammin [Nguyen et al., 2013].

SemFix käyttää tilastollista virheen paikannusta luomaan listan ohjelman lausekkeista, jotka on arvioitu virheen epäiltävyyden mukaan. Sen jälkeen SemFix aloittaa epäiltävimmästä lausekkeesta ja yrittää korjata sitä. Lausekkeen korjausyritys alkaa korjausrajoituksen luonnilla, jonka onnistuneen korjauksen on toteutettava. Tämän jälkeen SemFix yrittää ratkaista korjausrajoituksen ohjelman syntetisoinnilla [Nguyen et al., 2013].

Esimerkki, jossa käytetään otetta Tcas-ohjelmasta liikenteen törmäyksenvälttämis-järjestelmästä, selventää SemFixin toimintaa. Ohjelmakoodin ote on esitetty taulukossa 16. Oletetaan, että muuttujalla `inhibit` on vain kaksi sallittua arvoa, 0 ja 1. Ohjelman tarkoitettu käyttäytyminen on kuvailtu seuraavasti: `is_upward_preferred = (inhibit*100 + up_sep > down_sep)`. Testisarja, jolla testataan ohjelmaan virheettömyys, on esitetty taulukossa 17. Toteutus on virheellinen, koska kaksi testiä epäonnistuu. Virheen paikantamiseen SemFix käyttää aluksi Tarantula-virheenpaikannustyökalua annetulla testisarjalla. Tarantulan toimintaa on selvitetty luvussa 6. Tarantulan antamat epäiltävyydet, jotka on esitetty taulukossa 18, osoittavat rivin 4 olevan kaikista epäiltävin virheelliseksi, joten SemFix tutkii aluksi, onko virhe korjattavissa muuttamalla riviä 4 [Nguyen et al., 2013].

Rivi	Ohjelman rivi
1	`int is_upward_preferred(int inhibit, int up_sep, int down_sep) {`
2	`int bias;`
3	`if(inhibit)`
4	`bias = down_sep; //fix: bias=up_sep+100`
5	`else`
6	`bias = up_sep;`
7	`if (bias > down_sep)`
8	`return 1;`
9	`else`
10	`return 0;`
11	`}`

Taulukko 16. Esimerkki ohjelmakoodia SemFix-ohjelman toiminnan selittämiseksi. Koodi on ote Tcas-ohjelmasta.

Testi	Syöte			Odotettu tulos	Havaittu tulos	Testin Tulos
	inhibit	up sep	down sep			
1	1	0	100	0	0	Ok
2	1	11	110	1	0	Virhe
3	0	100	50	1	1	Ok
4	1	-20	60	1	0	Virhe
5	0	0	10	0	0	Ok

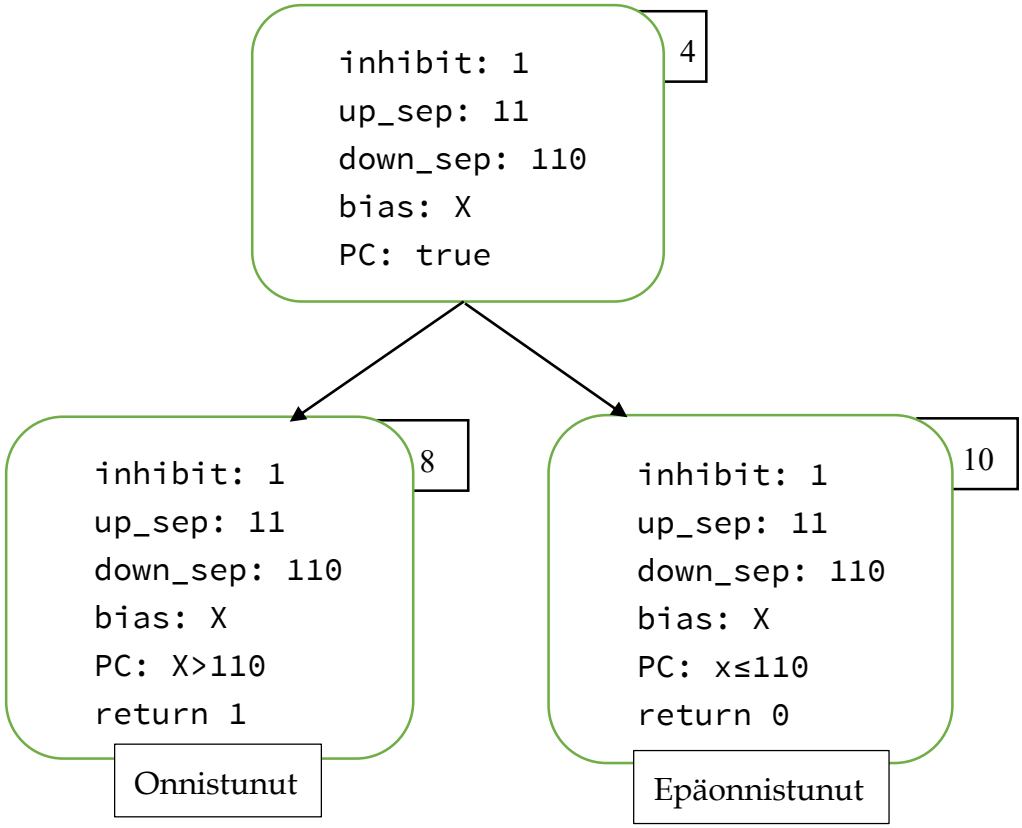
Taulukko 17. Testisarja Tcas-ohjelmalle ja testien tulokset.

Rivi	Epäiltävyys	Arvo
4	0.75	1
10	0.6	2
3	0.5	3
7	0.5	3
6	0	5
8	0	5

Taulukko 18. Tarantula-virheenpaikannustyökalun arvot virheen todennäköisyydelle Tcas-ohjelman otteelle.

Jos oletetaan, että rivi 4 halutaan korvata lauseella `bias = f(…)`, jossa funktion `f(…)` selvittäminen jää SemFixin tehtäväksi. Rivin 4 näkyvyysalueella on neljä muuttujaa, `inhibit`, `up_sep`, `down_sep` ja `bias`. Koska muuttujaa `bias` ei ole alustettu, oletetaan, että sitä ei voi käyttää korjauksessa. Seuraavaksi oletetaan funktion `f(…)` olevan muotoa `int f(int infibit, int up_sep, int down_sep)`. Nyt täytyy löytää rajoite, jonka funktion `f(…)` on täytettävä

kaikkien testisarjassa olevien testitapausten suorittamiseksi. Rajoite saadaan symbolisen suorittamisen kautta [Nguyen et al., 2013].



Kuva 2. Symbolisen suorituksen polku taulukon 17 testissä 2. Koodilaatikoiden ulkopuolella olevien suorakaiteiden sisällä oleva numero on ohjelmakoodin rivinumero. Polun tilaa kuvaa laatikoissa PC (Path Condition).

Jokaiselle testille, joka suorittaa rivin 4 luodaan funktiolle $f(\ldots)$ yksi rajoite, jonka toteutuminen takaa, että korjattu ohjelma tuottaa odotetun tuloksen. Käytetään taulukon 17 testiä 2 esimerkkinä rajoitteen luomiselle. Kuva 2 näyttää symbolisen suorituksen puurakenteen testille 2 syöttövektorilla (1, 11, 110), kun ohjelma on saavuttanut rivin 4. On huomattava, että esimerkissä symbolinen suorittaminen ei ala ohjelman syötteestä, vaan ohjelma on ajettu konkreettisesti, kunnes saavutetaan rivi 4. Tämän jälkeen muuttujan `bias` arvo korvataan symbolisella arvolla X ja suoritus jatkuu symbolisesti. Suorittaessaan haarautumista rivillä 7 suoritus kohtaa kaksi valintaa, jotka molemmat suoritetaan kuten näkyy symbolisessa suorituspuussa kuvassa 2. Testeistä tiedetään, että odotettu paluuarvo syötteelle (1, 11, 110) pitäisi olla 1 ja siten vain polkua, joka kulkee rivin 8 läpi, olisi seurattava, jotta ohjelma suorittaisi testin. Tämän polun kulkeminen vaatii ehdon X>110 täyttämistä. Ohjelman tila rivillä 4 on {`inhibit==1, up_sep==11, down_sep==110`}, jolloin funktion $f(\ldots)$ on täytettävä ehto $f(1,11,110)>110$. Samoin tiedetään, että $f(1,0,100)\leq 100$ testistä 1 ja $f(1,-20,60)>60$ testistä 4. Siten rajoite, jonka funktion $f(\ldots)$ on toteutettava on $f(1, 11, 110) > 110 \wedge f(1, 0, 100) \leq 100 \wedge f(1,-20, 60) > 60$. SemFix käyttää ohjelman syntetisointia rajoitteen ratkaisemiseksi funktiolle $f(\ldots)$, jolla konkreettinen funktio saadaan. Ohjelman syntetisointi vaatii peruskomponentteja (esimerkiksi vakioita sekä + ja -) aineksina funktion rakentamiseen. SemFix

antaa näitä komponentteja inkrementaalisesti ohjelman syntetisoinnille. Esimerkin ensimmäisessä yrityksessä vain vakio on saatavilla. Kuitenkaan mikään vakiofunktio ei toteuta rajoitetta $f(1, 11, 110) > 110 \wedge f(1, 0, 100) \leq 100 \wedge f(1,-20, 60) > 60$. Seuraavaksi funktion sallitaan käyttää yhtä +-operaatiota, jolloin f voi ottaa muodon `var1+c` tai `var1+var2`, missä `c` on kokonaislukuvakio ja `var1` ja `var2` ovat muuttujia jotka saavat arvonsa paramatreista `{inhibit,up_sep,down_sep}`. Nyt syntetisointimenettely löytää ratkaisun `f(int infibit, int up_sep, int down_sep)=up_sep+100`, joka on onnistunut korjaus Taulukko 16 olevaan ohjelmaan. On huomattava, että myös - -operaation käyttö mahdollistaisi oikean ratkaisun, joka olisi tällöin `f(int infibit, int up_sep, int down_sep)=up_sep-(-100)` [Nguyen et al., 2013].

Ohjelman korjauksen luonti tehdään komponenttipohjaisella ohjelman syntetisoinnilla. SemFixin kehittäjät käyttävät pitkälti samanlaisia kaavoja kuin luvussa 3 on käytetty. Pieniä eroja löytyy ja lisäksi syntetisoitavan ohjelman spesifikaatiota ei ole, joten SemFix paikkaa spesifikaation puuttumisen testitapauksien avulla.

Syntetisoinnille annetaan joukko syöte-tuloste -pareja, joista ohjelman syntetisointi tuottaa ohjelman, joka täyttää kaikkien annettujen syöte-tuloste -parien vaatimukset. Eli tarkemmin ilmaistuna, jos syöte-tuloste -pari on (α,β), niin syötteellä α syntetisoitu ohjelma tuottaa tulosteen β. Ohjelman syntetisointiin annetaan joukko peruskomponentteja, joita syntetisoitavan funktion f sallitaan käyttää. Esimerkiksi jos ohjelma syntetisoidaan lineaarisilla lausekkeilla, annettujen peruskomponenttien joukko on {vakio, plus, miinus} [Nguyen et al., 2013].

Jokaiselle komponentille on määritelty joukko paikkamuuttujia. Syntetisointiprosessi on pelkistetty löytämään arvot näille komponenttien paikkamuuttujille. Paikkamuuttujien arvoja rajoittava rajoite (*constraint*) on ensimmäisen kertaluvun logiikan kaava ja ratkaistaan SMT-ratkaisijalla. Jos SMT-ratkaisija löytää ratkaisun eli arvot paikkamuuttujille, voidaan saatujen paikkamuuttujien arvojen pohjalta rakentaa yksikäsitteinen ohjelma [Nguyen et al., 2013]. Lisään huomautukseksi että yksikäsitteinen ohjelma ei nyt tarkoita, ettei ohjelmaa voisi tehdä muulla tavalla, vaan yksikäsitteisyys tarkoittaa syötteiden ja tulosteiden vastaavuutta käytettävissä olevilla komponenteilla.

Oletetaan esimerkiksi, että annetaan N komponenttia $\{f_1,\ldots,f_N\}$ funktion f syntetisoimiseksi. Oletetaan lisäksi, että jokainen komponentti tuottaa vain yhden tulosteen. Merkitään i:nettä komponentin syötettä merkinnällä $\vec{X}_i$ ja tulostetta vastaavasti merkinnällä r_i. Merkitään kaikkien komponenttien syöttömuuttujien joukkoa Q:lla ja kaikkien komponenttien tulostetta R:llä. [Nguyen et al., 2013]. Matemaattisesti tämä voidaan esittää muodossa

$$Q := \cup_{i=1}^{N} \vec{X_i}, R := \cup_{i=1}^{N} \{r_i\}.$$

Kaavoissa $\vec{X}$ tarkoittaa funktion f syötemuuttujia ja r funktion f tulostemuuttujaa. Paikkamuuttujien joukko L on $L := \left\{ l_x \middle| x \in Q \cup R \cup \vec{X} \cup \{r\} \right\}$. Paikkamuuttuja l_x osoittaa, missä muuttuja x on määritelty [Nguyen et al., 2013].

Antamalla joukon L paikkamuuttujille arvot ohjelma voidaan rakentaa menettelyllä *Lval2Prog(L)*. Tässä rakennetun ohjelman i:s rivi on $r_j = f_j(r_{\sigma(1)}, \ldots, r_{\sigma(\eta)})$, kun $l_{r_j} == i$ ja $\wedge_{k=1}^{\eta} (l_{x_j^k} == l_{r_{\sigma(k)}})$, missä η on syötteiden määrä komponentille f_j ja x_j^k tarkoittaa komponentin f_j k:nnetta syöttöparametria. Ohjelman tuloste on tuotettu riville l_r [Nguyen et al., 2013].

Vaikka luvussa 3 on oma esimerkkinsä paikkamuutujien merkityksen selittämistä varten, olen katsonut aiheelliseksi esittää myös SemFixin kehittäjien esimerkin. Tällä kertaa käytän erilaista esitystapaa paikkamuuttujien tulkintaa varten, joka toivottavasti auttaa asian selventämisessä. Syntetisoitu esimerkki ja paikkamuuttujien tulkintaesitys on esitetty taulukossa 19. Oletetaan esimerkiksi, että annetaan N komponenttia $\{f_1,\ldots,f_N\}$ funktion f syntetisoimiseksi. Oletetaan, että saatavilla on vain yksi komponentti +, jonka syötteet ovat x_+^1 ja x_+^2. Selkeyden vuoksi, koska komponenttejakin on vain yksi eli +, käytetään alaindeksinäkin vain merkintää +. Jos komponentteja olisi enemmän, kannattaisi käyttää numeroindeksiä. Tulostemuuttuja +:lle on r_+. Oletetaan, että syntetisoidulle ohjelmalle on vain yksi syöte. Annetaan paikkamuuttujille arvot $l_{r_+} == 1$, $l_{x_+^1} == 0$, $l_{x_+^2} == 0$, $l_r == 1$, $l_{x^1} == 0$. Koska $l_{r_+} == 1$, niin r_+ on määritelty rivillä 1 ja siten komponentti + sijoitetaan riville 1. Jos paikkamuuttujat $l_{x_+^1} == l_{x_+^2} == 0$, niin silloin molemmat x_+^1 ja x_+^2 ovat samoja kuin muuttuja määritettynä rivillä 0, mikä tarkoittaa, että ne ovat samoja kuin rivin 0 tuloste. Koska l_r on yhtäsuuri kuin 1, niin arvo määriteltynä rivillä 1, r_+, on ohjelman tuloste. Paikkamuuttujien arvoista voidaan rakentaa taulukossa 19 esitetty ohjelma, missä `input`^{0} merkitsee syntetisoidun ohjelman ensimmäistä syöteparametria [Nguyen et al., 2013].

	x-indeksi	1	2	3	(Ei paikkamuuttujaa)	(Ei paikkamuuttujaa)
i:s rivi.		Kirjaston komponentin syötteet		Kirjaston komponentin tuloste	Luodun funktion syöte	Luodun funktion tuloste
$l_{r_j}==i$	r_j eli luodun funktion ohjelmakoodi	$l_{x_+^1}$	$l_{x_+^2}$	l_{r_+}	l_{x^1} `input`^{0}	l_r
0	`r`$_0$`=input`^{0}`;`	Q	Q		$\vec{X}$	
1	`r`$_+$`=r`$_0$`+r`$_0$`;`			R		$\{r\}$, Syöte funktion tulosteelle.
2	`return r`$_+$`;`					

Taulukko 19. Esimerkki ja toisenlainen esitystapa paikkamuuttujien tulkinnasta.

Paikkamuuttujien täytyy täyttää tiettyjä rajoitteita, jotta vastaava ohjelma voi läpäistä kaikki annetut syöte-tuloste -parit. Ensimmäinen on hyvinmuodostuneisuusrajoite (*well-formedness*) ψ_{wfp}, joka on muotoa

$$\psi_{wfp}(L,Q,R) \stackrel{\text{def}}{=} \bigwedge_{x\in Q}(0 \le l_x < M) \wedge \bigwedge_{x\in R}(|\vec{x}| \le l_x < M)$$
$$\wedge\, \psi_{cons}(L,R) \wedge \psi_{acyc}(L,Q,R),$$

jossa johdonmukaisuusrajoite aukikirjoitettuna on

$$\psi_{cons}(L,R) \stackrel{\text{def}}{=} \bigwedge_{x,y\in R, x\neq y} (l_x \neq l_y),$$

ja asyklisyysrajoite aukikirjoitettuna

$$\psi_{acyc}(L,Q,R) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{N} \bigwedge_{x\in \vec{X}_i, y\equiv r_i} (l_x < l_y).$$

Kaavoissa $M = |\vec{x}| + N$, missä N on ohjelman syntetisointimenettelyyn saatavien komponenttien lukumäärä. Rajoite ψ_{cons} määrää, että jokaisella rivillä on vain yksi komponentti, ja ψ_{acyc} määrää, että jokaisen komponentin syötteet ovat määriteltyjä ennen kuin niitä käytetään [Nguyen et al., 2013].

Rajoite ψ_{wfp} takaa vain, että joukon L arvot vastaavat hyvinmuodostunutta ohjelmaa f. Antamalla joukon syöte-tuloste -pareja komponenttipohjainen ohjelman syntetisointi tuottaa ohjelman, joka täyttää kaikkien annettujen syöte-tuloste -parien vaatimukset. Jos syöte-tuloste -pari on (α,β), niin syötteellä α syntetisoidun ohjelman tulisi antaa tuloste β. Jotta ratkaisu f toteuttaa kaikki syöte-tuloste –parit tarvitaan takaamaan lisäksi seuraava rajoite

$$\phi_{func}(L,\alpha,\beta) \stackrel{\text{def}}{=} \psi_{conn}\left(L,\vec{X},r,Q,R\right) \wedge \phi_{lib}(Q,R) \wedge \left(\alpha = \vec{X}\right) \wedge (\beta = r),$$

jossa

$$\phi_{lib}(Q,R) \stackrel{\text{def}}{=} \left(\bigwedge_{i=1}^{N} \phi_i\left(\vec{X}_i, r_i\right)\right),$$

ja liitettävyysrajoite

$$\psi_{conn}\left(L,\vec{X},r,Q,R\right) \stackrel{\text{def}}{=} \bigwedge_{x,y\in Q\cup R\cup\vec{X}\cup\{r\}} \left(l_x = l_y \Rightarrow x = y\right).$$

Vertaa rajoitetta luvun 3 tarkistusrajoitteeseen

$$\forall \vec{I}, O, P, R: \left(\phi_{lib}(P,R) \wedge \phi_{conn}\left(\vec{I},O,P,R\right)\right) \Rightarrow \phi_{spec}\left(\vec{I},O\right).$$

Tarkistusrajoitteessa komponenttien syötteiden joukko on merkitty kirjaimella P. Liitettävyysrajoitteessa on käytetty versiota ϕ_{conn} version ψ_{conn} sijaan. Myös syntetisointirajoite

$$\exists L \forall \vec{I}, O, T: \psi_{wfp}(L) \wedge \left(\phi_{lib}(T) \wedge \psi_{conn}\left(\vec{I},O,T,L\right) \Rightarrow \phi_{spec}\left(\vec{I},O\right)\right)$$

on hyvä muistaa. Tässä rajoitteessa joukot P ja R on yhdistetty joukoksi T ja käytetään liitettävyysrajoitteessa ψ_{conn} versiota.

Jokaisen peruskomponentin semantiikka on enkoodattu kirjastoon ϕ_{lib}, ja ϕ_i esittää komponentin f_i spesifikaatiota. Paikkamuuttujien ja ohjelman muuttujien välinen suhde on enkoodattu liitettävyysrajoitteeseen ψ_{conn}. Yhteenvetona $\phi_{func}(L,\alpha,\beta)$ esittää, että suorittaessa syntetisoitua funktiota f syötteellä α_i, tulosteen pitäisi olla β_i [Nguyen et al., 2013]

$$\theta \stackrel{\text{def}}{=} (\bigwedge_{i=1}^{n} \phi_{func}(L, \alpha_i, \beta_i)) \wedge \psi_{wfp}(L, Q, R).$$

Lopuksi rajoite θ esittää, että annettuna *n* syöte-tuloste -paria $\{(\alpha_i, \beta_i) | 1 \leq i \leq n\}$, syntetisoidun funktion *f* tulisi täyttää kaikkien syöte-tuloste -parien ehdot ja että funktion täytyy olla hyvinmuodostettu. Annetulla ratkaisulla L_0 rajoitteelle θ, voidaan rakentaa ohjelma, joka täyttää ehdot kaikille syöte-tuloste -pareille (α_i, β_i) käyttämällä menetelmää *Lval2Prog* [Nguyen et al., 2013].

SemFix tekee vain yhtä lauseketta muuttavia korjauksia. Korjauksissa ei korjata sijoituksissa lauseen vasenta puolta, eli muuttujaa, johon sijoitetaan, ei muuteta. Korjausten muoto sijoituksissa on

```
x = fbuggy(...) -> x = f(...).
```

Korjausten muoto haarautumislausekkeissa on puolestaan

```
if(fbuggy(...)) -> if(f(...)).
```

Kummassakin muodossa luodaan ilmaisu `f(...)`, jota käytetään korvaamaan sijoituslauseen oikea puoli tai haarautumislauseen predikaatti. Luotu ilmaisu `f(...)` ei saa aiheuttaa sivuvaikutuksia. Esimerkiksi luotu ilmaisu `f(...)` ei muuta ohjelman muuttujien sisältöä, kun sitä suoritetaan. Korjaukset voivat silti olla ei-triviaaleja kuten seuraava esimerkki erään TCAS-ohjelman version korjauksesta osoittaa. SemFix kehitti korjauksen `tmp = ((Other_Capability<Alt_Layer_Value)?Two_of_Three_Reports_Valid:Cur_Vertical_Sep);` lauseelle `tmp = Up_Separation` [Nguyen et al., 2013].

Ilmaisu on pohjimmiltaan funktio. Esimerkiksi ilmaisu `x+y` voidaan käsitellä funktiona `f(x,y)=x+y` [Nguyen et al., 2013].

Korjausrajoite (*repair constraint*) voidaan määritellä seuraavasti: Annetulle ohjelmalle P, testisarja T, funktion f_{buggy} ohjelmassa P korjausrajoite C on sellainen toteutuva rajoite funktiolle f, että jos $f \vDash C$, P[f_{buggy}/f] suorittaa hyväksytysti kaikki testit testisarjassa T [Nguyen et al., 2013].

Korjausrajoite C on testisarjasta T johdettujen rajoitteiden konjunktio eli yhdiste. Oletetaan, että testisarjassa on *n* testiä, $T = \{t_i \mid 1 \leq i \leq n\}$, jotka oikeasti ovat eri syöttövektoreita ajettaessa ohjelmaa P. Korjausrajoite on $C = \wedge_{i=1}^{n} C_i$. Jokaiselle testille t_i luodaan vastaava korjausrajoite C_i, jonka ehdot täyttää korjattu funktio f siten, että jos ohjelman virheellinen funktio f_{buggy} korvataan funktiolla f, ohjelma suorittaa hyväksytysti testin t_i [Nguyen et al., 2013].

Jokainen rajoite C_i on funktion f toisen kertaluvun predikaatti. Rajoitteen C_i luomiseen käytetään symbolista suorittamista. Semfix:in kehittäjien symbolinen suorittaminen alkaa korjattavasta lauseesta *s*, johon asti ohjelma on suoritettu konkreettisesti konkreettisella syötteellä t_i. Lausetta *s* ei suoriteta konkreettisesti. Ohjelman tila merkitään ξ_i ennen lauseen *s* suorittamista. Funktion `f(...)` tulos asetetaan symboliseksi, merkitään τ_i, ja jatketaan symbolista suorittamista lauseesta *s*. Oletetaan, että symbolinen suorittaminen tutkii *m* polkua. Jokaista tutkittua polkua π_j, $1 \leq j \leq m$, kohden liittyvä polun tila merkitään pc_j ja symbolinen ilmaisu tulosteelle O_j. Ohjelman *P* odotettua tulostetta syötteellä t_i merkitään $O(t_i)$. Rajoite C_i on

$$C_i \coloneqq (\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))) \wedge (f(\xi_i) == \tau_i).$$

Ensimmäinen osio rajoitteesta C_i, $\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))$, takaa, että on ainakin yksi mahdollinen polku (kuten taattu polun tilan toteutuvuudella) sellaisen ohjelman P tulosteen kanssa, joka on sama kuin odotettu tuloste $O(t_i)$. Toinen osio $(f(\xi_i) == \tau_i)$, muodostuu funktion f syöte-tuloste -parien yhteyksistä. Kun ohjelman tila ξ_i on syötteenä, funktion f tulosteen τ_i täytyy toteuttaa $\bigvee_{j}^{m}(pc_j \wedge O_j == O(t_i))$. Silloin, kun lausetta s ei läpikäydä ohjelman P suorituksen aikana syötteellä t_i, C_i on tosi, jos suoritus läpäisee ja epätosi jos suoritus epäonnistuu.

Rajoitteen C_i luonti esimerkkinä annetun taulukon 16 ohjelmalle käyttäen toista testiä taulukosta 17. Symbolisen suorituksen polun puurakenne testille on esitetty kuvassa 2. Merkintää X käytetään symboliselle muuttujalle. Symbolinen suoritus tutkii kahta polkua. Ensimmäisen polun tila on X>110 ja tuloste on 1. Toisen polun tila on X≤110 ja tuloste on 0. Koska testin odotettu tulos on 1, niin C_i:n ensimmäinen osa on (X>110 ∧ 1=1) ∨ (X≤110 ∧ 0=1) joka voidaan yksinkertaistaa X>110. Ohjelman tila korjattavassa kohdassa on `{inhibit==1, up_sep==11, down_sep==110}`, jolloin toinen osa C_i:stä on f(1,11,110)=X. Kokonaisuudessaan C_i on siten f(1,11,110)>110.

Kun rajoite C_i on luotu jokaisesta testistä t_i, saadaan kokonainen rajoite C, joka on

$$C \coloneqq \bigwedge_{i=1}^{n}((\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))) \wedge (f(\xi_i) == \tau_i)).$$

Rajoitteen C ratkaisu on funktio f jota voidaan käyttää ohjelman korjaamiseen siten että testisarja T:n testit suoriutuvat lävitse. Rajoite C on kuitenkin liian vaikea nykyisille SMT-ratkaisijoille suoraan ratkaistavaksi. Jos funktio f on vakiofunktio, niin silloin rajoite C tässä muodossa voidaan ratkaista. Tässä tapauksessa $f(\ldots)$ voidaan korvata vapaalla muuttujalla ja rajoite C muuttuu ensimmäisen kertaluvun rajoitteeksi, joka on ratkaistavissa saatavilla olevilla SMT-ratkaisijoilla [Nguyen et al., 2013].

Jos korjattava lause suoritetaan useammin kuin kerran, joko silmukan sisällä tai useamman kerran kutsuttavan funktion sisällä, testin t_i suorituksen aikana, käytetään merkintää τ_i^k esittämään $f(\ldots)$ tuottamaa arvoa, kun se suoritetaan k:tta kertaa. Merkintää ξ_i^k käytetään esittämään ohjelman tilaa ennen kuin f suoritetaan k:tta kertaa, kun ohjelma P suoritetaan syötteellä t_i. Huomioi, että ξ_i^k:ssa olevien muuttujien arvot voidaan esittää symbolisesti muuttujilla $\{\tau_i^k,\ldots,\tau_i^{k-1}\}$. Oletetaan, että lause s suoritetaan w kertaa symbolisen suorituksen aikana. Tällöin saadaan korjausrajoite

$$C_i \coloneqq (\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))) \wedge (\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k)$$

ja

$$C \coloneqq \bigwedge_{i=1}^{n} C_i.$$

Nyt jokainen symbolinen tuloste O_j on muuttujien ilmaisu $\tau_i^1,\ldots,\tau_i^w$. Toinen osa C_i:stä muuttuu muotoon $(\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k)$, mikä merkitsee, että funktion f syöte-tuloste –suhteiden täytyy

toteutua joka kerran, kun f suoritetaan. Silloin, kun f suoritetaan k:tta kertaa, ohjelman tilalla ξ_i^k, funktion f tuloste on τ_i^k [Nguyen et al., 2013].

Päättymättömän silmukan tapauksessa, kun silmukan lopettamisen tila on ilmaisu esiteltyjen symbolisten muuttujien avulla, symbolinen suorittaminen ei välttämättä pääty koskaan tutkien äärettömän määrän silmukoiden iteraatioita. Otetaan esimerkiksi `while(i<x){ fx=buggy-expression;}`. Aina kun virheellinen lauseke suoritetaan, sijoitetaan uusi symbolinen arvo muuttujaan x. Tarkistettaessa silmukan tilaa `i<x` seuraavalle iteraatiolle, molemmat vaihtoehdot ovat mahdollisia, koska uusi symbolinen x:n arvo on rajoittamaton. Siten symbolinen suoritus jatkaa tutkimista lisääntyviä silmukka-iteraatioita eikä koskaan pääty. Tämän äärettömän silmukoiden tutkimisten estämiseksi SemFix:in kehittäjät ottivat käyttöön silmukkarajan B jokaiselle silmukalle symbolisessa suorituksessa. Kun silmukka on toistettu B-kertaa, symbolinen suoritus lopettaa etsimästä polkua, joka johtaisi uuteen toistoon. Tämä ei kuitenkaan riko korjausrajoitteen kelpoisuutta. Jos korjausrajoitteella on ratkaisu f ali-rajoitteelle C_i, yksi rajoitteista $pc_j \wedge O_j == O(t_i)$ on toteutunut C_i:ssä. On taattua, että kun käytetään korjausta f, korjattu ohjelma syötteellä t_i, tulee seuraamaan polkua π_j, jonka polun tila on pc_j ja luo oikean tulosteen. Haittapuolena on, jos silmukka raja B on liian pieni, että ei havaita korjauksia, jotka ohjaisivat ohjelmaa seuraamaan polkuja, jotka sisältäisivät suuremman määrän silmukoiden toistoja [Nguyen et al., 2013].

Seuraavaksi tarkastellaan korjausrajoitteen ratkaisua. Syntetisoitavan ohjelman syöte-tuloste -parit on enkoodattu mukaan rajoitteisiin paikkamuuttujien joukolle L, jonka arvottaminen johtaa ohjelmaan, joka toteuttaa kyseiset syöte-tuloste -parit. Rajoite $\psi_{func}(L, \alpha, \beta)$ määrää, että syntetisoitu ohjelma tuottaa tulosteen β syötteellä α. Korjausrajoitteessa on myös syöte-tuloste –pari $\langle \xi_i^k, \tau_i^k \rangle$, joka on generoitu, kun f on suoritettu k:tta kertaa ohjelman P ajolla syötteellä t_i. Kuitenkin $\langle \xi_i^k, \tau_i^k \rangle$ on symbolinen muuttujien $\{\tau_i^k | 1 \leq k \leq w\}$ lauseke, missä w on niiden toistojen määrä, kun f suoritettu syötteellä t_i. Lisäksi muuttujien $\{\tau_i^k | 1 \leq k \leq w\}$ täytyy toteuttaa ehto $(\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i)))$, jotta suoritus läpäisee testin eli tuloste on odotettu. Siten rajoite, joka f:n pitää toteuttaa suorittaessa syötettä t_i ,on

$$\theta \stackrel{\text{def}}{=} \exists \vec{\tau_i}, \bigwedge_{k=1}^{w} \phi_{func}(L, \xi_i^k, \tau_i^k) \wedge (\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))),$$

missä $\tau_i \coloneqq \{\tau_i^k | 1 \leq k \leq w\}$ [Nguyen et al., 2013].

Yhdistämällä rajoitteet kaikista testeistä hyvinmuodostuneisuusrajoitteen ψ_{wfp} kanssa saadaan rajoite θ. Se on ratkaisu joka vastaa funktiota f, mikä on kelvollinen korjaus. Rajoitteen kaava on

$$\theta \stackrel{\text{def}}{=} (\bigwedge_{i=1}^{n} \theta_i) \wedge \psi_{wfp}(L),$$

josta SemFix:in kehittäjät ovat tehneet optimointeja [Nguyen et al., 2013].

Eräs optimointi on vain rajoitteen sen osan toteutuvuuden tarkastelu, joka on saatu symbolisella suorituksella. Tällöin tarkastellaan korjausrajoitteen C osarajoitteen

$$C_i \coloneqq (\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))) \wedge (\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k)$$

alkuosaa. Olkoon tämä tarkasteltava alkuosa nyt merkitty merkinnällä

$$C_i' \coloneqq (\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i))),$$

joka on symbolisten muuttujien $\{\tau_i^k | 1 \leq k \leq w\}$ ensimmäisen kertaluvun rajoite, missä w on symbolisten muuttujien lukumäärä. Jokaisen rajoitteen C_i' toteutuvuus tarkistetaan Z3-SMT-ratkaisijalla. Jos yksikään C_i' ei ole toteutuva, niin myöskään C_i ei ole toteutuva ja siten korjausrajoite C ei ole toteutuva. Eli polun tila pc_j ja symbolinen ilmaisu tulosteelle O_j eivät vastaa odotettua tulostetta $O(t_i)$. Tällöin päätellään, että lause ei ole korjattavissa ja ohjelman syntetisointia ei suoriteta ollenkaan [Nguyen et al., 2013].

Semfix:in algoritmi, joka käyttää tässä kohdassa aiemmin esiteltyjä tekniikoita on esitetty taulukossa 20. Algoritmi tutkii listan virheellisiksi epäiltävistä ohjelman lauseista RC iteratiivisesti kunnes onnistunut korjaus pystytään luomaan. Jokaisella iteraatiolla listasta RC otetaan kaikista epäilyttävin lause, jota ei vielä ole kokeiltu kutsulla Shift(RC). Koska testisarja voi olla laaja, se voi aiheuttaa skaalautuvuus ongelmia, käytetään testisarjasta alijoukkoa (*subset*) korjauksen luontiin. Testien valintaa ohjataan testaamisella. Joukkoa S käytetään säilyttämään testejä, joita käytetään korjausrajoitteen generointiin. Aluksi joukossa S ovat vain T:n epäonnistuneet testit. Korjauksen luonnin jälkeen testataan epäonnistuuko mikään testi, joka ei ole joukossa S korjatussa ohjelmassa. Jos testi t' epäonnistuu korjatussa ohjelmassa P', testi t' lisätään joukkoon S ja korjaus luodaan uudelleen. Tämä korjausten generointi päättyy varmasti, koska joukon S maksimikoko on rajoitettu testisarjalla T [Nguyen et al., 2013].

1	Input
2	P: The buggy program
3	T: A test suite
4	RC: A ranked list of potential bug root-cause
5	Output: r: A repair for P
8	while RC is not EMPTY and not TIMEOUT do
9	rc = Shift(RC) // A repair candidate
10	S = ∅ // A test suite for repair generation
11	T_f = ExtractFailedTests(T, P);
12	while $T_f \neq \emptyset$; do
13	S = S ∪ T_f
14	new repair = Repair(P, S, rc)
15	if new repair == null then
16	break
17	end if
18	P' = ApplyRepair(P, new repair)
19	T_f = ExtractFailedTests(T, P');
20	end while
21	if new repair not null then
22	return new repair
23	end if
24	end while
25	
26	function Repair(P, S, rc)
27	C = GenerateRepairConstraint(P, S, rc);
28	level = 1 // The complexity of a repair
29	new repair = Synthesize(C, level);
30	while new repair == null and level ≤ MAX LEVEL do
31	level = level + 1
32	new repair = Synthesize(C, level);
33	end while
34	return new repair
35	end function

Taulukko 20. SemFix:in algoritmi ohjelman korjaamiseksi

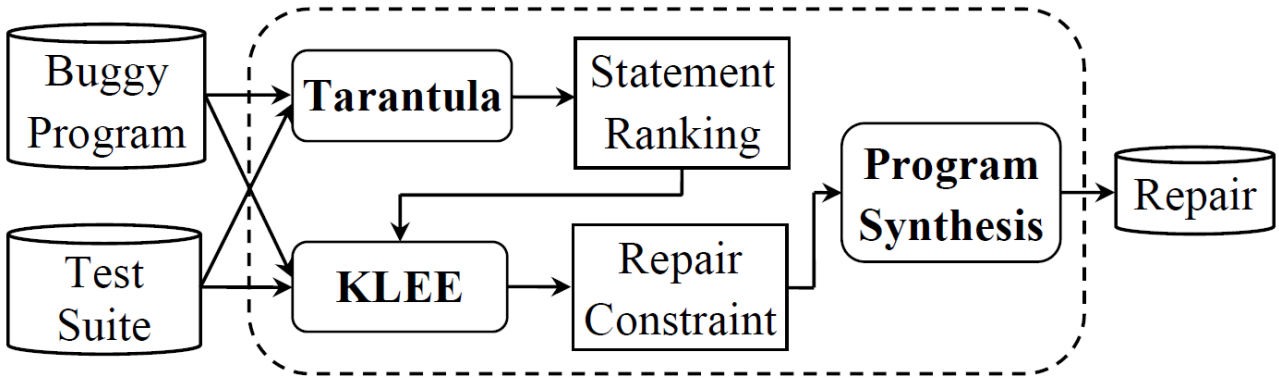
Korjauksen syntetisoinnin aikana SemFix antaa syntetisoinnin käytettäväksi joukon komponentteja, joita voidaan käyttää korjaukseen. Ohjelman syntetisoinnin skaalautuvuuden parantamiseksi ja luotujen korjausten monimutkaisuuden vähentämiseksi yleiset syntetisoinnissa

käytetyt komponentit on lajiteltu niiden monimutkaisuuden mukaan ja niitä lisätään syntetisointi proseduurille vähitellen. Komponentit on lueteltu taulukossa 21 [Nguyen et al., 2013].

Level	Conditional Statement	Assign Statement
1	Constants	Constants
2	Comparison (>, ≥, =, ≠)	Arithmetic (+, -)
3	Logic (∧, ∨)	Comparison, Ite
4	Arithmetic (+, -)	Logic
5	Ite, Array Access	Array Access
6	Arithmetic (*)	Arithmetic (*)

Taulukko 21. SemFix:ssä käytetty ohjelmien syntetisointiin käytettävien yleisten komponenttien monimutkaisuuden luokittelu.

Ensimmäisessä tasossa vain vakiot ovat sallittuja tehdyssä korjauksessa. Jos korjaus epäonnistuu, lisätään komponentteja asteittain. Prosessi jatkuu, kunnes korjaus on generoitu tai `MAX_LEVEL`-arvo on saavutettu. SemFix-automatiikkaa rajoitetaan 4 tasosta ylöspäin. Näitä komponentteja korjauksen luontiin voidaan käyttää vain, jos käyttäjä antaa SemFix:lle virheiden sijainnit [Nguyen et al., 2013].



Kuva 3. SemFixin komponentit ja niiden välillä siirtyminen [Nguyen et al., 2013].

SemFix-ohjelman komponentit on esitelty kuvassa 3. Suoritus etenee niin, että virheellinen ohjelma ja testisarja ovat aluksi syötteenä Tarantulalle, virheiden paikantamisohjelmalle. Tarantula luo arvioidun ohjelman lauseiden virheen todennäköisyyksistä ja tämä lista ja testisarja ovat seuraavaksi syötteinä KLEE-työkalulle. KLEE on symbolisen suorittamisen hoitava työkalu, ja se tuottaa syötteidensä avulla korjausrajoitteen. Korjausrajoite on syötteenä ohjelman syntetisoinnille jossa on kaksi vaihetta. Rajoitteen ratkaiseminen, jossa käytetään Z3 SMT-ratkaisijaa ja PHP-kielellä toteutettu koodin generointi, joka tuottaa lopullisena tulosteena korjauksen [Nguyen et al., 2013].

SemFixin kehittäjät testasivat SemFixiä 90 virheellisen ohjelman kanssa. SemFix ei onnistunut korjaamaa 42:ta virhettä ja SemFixin kehittäjät totesivat seuraavien seikkojen alentavan SemFixin korjaamien virheiden määrää: Liian alhainen silmukkaraja aiheutti kahdeksan epäonnistunutta korjausta. Kuudessa tapauksessa tarvittiin tarkkaa taulukoiden mallintamista, mitä testien aikainen versio SemFixistä ei vielä täysin tue. Viisi virhettä jäi korjaamatta, koska symbolinen suorittaminen

olisi vaatinut liukuluku-muuttujia ja käytetty työkalu KLEE ei tukenut niitä. Useamman kuin yhden rivin korjauksia tarvittiin 15 tapauksessa, ja SemFix pystyy vain yhden rivin korjauksiin. Tunnistamattomia syitä virheen korjaamattomuuteen oli kahdeksan [Nguyen et al., 2013].

	Testisarjan koko.									
	10		20		30		40		50	
Virheellinen ohjelma	SF	GP	SF	GP	SF	GP	SF	GP	SF	GP
Tcas	38	24	38	19	35	16	34	12	34	11
Schedule	5	1	3	1	4	1	4	0	4	0
Schedule2	4	4	3	2	4	2	3	3	2	1
Replace	7	6	7	5	8	5	7	6	6	4

Taulukko 22. Korjattujen virheiden määrä verrattuna testisarjan kokoon. Soluissa on korjattujen virheiden määrä. SF SemFix ja GP GenProg. Taulukko on typistetympi versio SemFixin kehittäjien laajemmasta taulukosta [Nguyen et al., 2013].

On myös hyvä huomioida, että korjauksella tarkoitetaan vain testisarjan läpäisevää muutosta. Siten on hyvin yleistä että testien lisääminen saattaa huonontaa korjattavien virheiden määrää. Tämän ilmiön voi todentaa taulukossa 22 olevista tuloksista korjatuista virheistä verrattuna testisarjan suuruuteen. On tietenkin selvää, että on vaikeampi tehdä korjaus, joka läpäisee useampia testejä. Siten korjaus ei välttämättä ole kelvollinen, kun testejä lisätään. Tämä heikkous koskee kaikkia menetelmiä, jotka kehittävät korjauksensa testisarjoihin perustuen [Nguyen et al., 2013].

5.3. Semanttinen analyysi DirectFix-työkalulla

DirectFix on semantiikkaan perustuva korjausmenetelmä, jonka tarkoituksena on kehittää yksinkertaisin korjaus virheelle. Tällöin ohjelman alkuperäinen rakenne halutaan säilyttää ja korjattavia rivejä halutaan olevan mahdollisimman vähän. DirectFix:ssä virheen paikantaminen ja korjaus tapahtuu samalla kerralla. DirectFix aiheuttaa kehittäjien mukaan myös vähemmän regressiota kuin SemFix. Tätä väitettä voinee pitää luotettavana, koska kummatkin on kehitetty samassa yliopistossa ja yksi kehittäjistä on ollut osallisena kummankin kehittämisessä [Mechtaev et al., 2015].

DirectFix:n kehittäjät ovat havainneet, että korjauksen yksinkertaisuuteen vaikuttaa korjauksen sijainti. Testi-ohjatut korjausmenetelmät tukeutuvat tilastollisiin virheen paikantimiin, joissa virheen todennäköistä sijaintia arvioidaan epäiltävyydellä. Epäiltävyydellä ja virheen korjauksen yksinkertaisuudella ei kuitenkaan ole suoraa verrannollisuutta. Siksi virheen paikantaminen ja korjausvaiheet on yhdistetty käyttäen osittaista MaxSAT ratkaisemista siten, että tämä yhdistetty menetelmä olisi myös skaalautuva. Täten pääasiallinen idea on välttää korjauksen kehittäminen kaikkiin niihin kohtiin, johon se on mahdollista tehdä ja sen jälkeen valita korjauksista yksinkertaisin,

koska tämä olisi liian tehotonta. Tämä saavutetaan pelkistämällä ongelma maksimaaliseksi toteutuvuusongelmaksi (*maximum satisfiability problem*) [Mechtaev et al., 2015].

Menetelmässä tehdään looginen kaava annetulle virheelliselle ohjelmalle ja testisarjalle sillä tavalla, että kaavan malli, eli sijoitukset muuttujiin toteuttavat kaavan, on yksinkertaisin korjaus. DirectFix käyttää omaa osittaista MaxSMT-ratkaisijaa, joka on toteutettu Z3-SMT-ratkaisijan päälle. Maksimaalinen toteutuvuusongelma on SAT-ongelman yleistys, jonka tarkoituksena on löytää maksimaalinen määrä klausuuleita, jotka voivat toteutua annetulle kaavalle. MaxSMT on samankaltainen yleistys SMT:lle. Osittainen maksimaalinen toteutuvuus pMaxSAT (*partial maximum satisfiability*) on sellaisten klausuulien maksimaalisen alijoukon s_{max} klausuuleista s löytämisen ongelma, että $s_{max} \wedge h$ on toteutuva, jossa s on heikkojen klausuulien joukko (*set of soft clause*) ja h on tiukkojen klausuulien joukko (*set of hard clauses*). Heikkojen klausuulien ei aina tarvitse toteutua. Tiukkojen klausuulien täytyy aina olla toteutuvia. SMT:lle vastaavasti pMaxSMT on pMaxSAT:in yleistys SMT:lle [Mechtaev et al., 2015].

DirectFix käyttää ohjelmien semantiikkoja, jotka ilmaistaan loogisen kaavan kautta, jota kirjallisuudessa kutsutaan jäljityskaavaksi *TF* (*trace formula*). Deterministisen ohjelman *P* jäljityskaava *TF* on looginen kaava, joka toteuttaa seuraavan ominaisuuden:

Annetulla syötteellä *I* ohjelmalle *P*, *TF*(*I*,*O*) on toteutuva, jos ohjelman *P* tuloste on *O*, ja ei toteutuva, jos tuloste ei ole *O*, jossa *TF*(*I*,*O*) tarkoittaa jäljityskaavaa *TF*, jonka syötemuuttuja on rajoitettu *I*:llä ja tulostemuuttuja *O*:lla [Mechtaev et al., 2015].

Testitapauksista saadaan enkoodattua ns. oraakkelirajoite komponenttipohjaista syntetisointia varten. Konponenttipohjainen syntetisointi yhdessä oraakkelirajoitteen kanssa toimii yhdellä testitapauksella. Jos yritetään yhdistää kaikki testitapaukset mukaan, enkoodattu kaava ei ole toteutuva. Enkoodaus saadaan laajennettua useammalle testitapaukselle uudelleen nimeämällä muuttujat jokaiselle testitapaukselle yksikäsitteisillä muuttujien nimillä. Siten kaava, joka kattaa kaikki syöte-tuloste -yhteydet, on jokaiselle testitapaukselle uudelleen nimettyjen kaavojen yhdistelmä [Mechtaev et al., 2015].

Virheellisestä ohjelmasta hankitaan oletetun virheellisen kohdan läheltä jäljityskaava. Jäljityskaava taulukon 23 virheelliselle funktiolle foo on

$\varphi_{buggy} \equiv (\text{if } (x_1 > y_1) \text{ then } (y_2 = y_1 + 1) \text{ else } (y_2 = y_1 - 1)) \wedge (\text{result} = y_2 + 2)$.

Jäljityskaavan muuttujat x_i ja y_i vastaavat taulukon 23 funktion foo muuttujia `x` ja `y`, ja result vastaa funktion foo paluuarvoa. Samassa taulukossa on testausfunktio test_foo, joka epäonnistuu. Tässä esimerkissä on yksinkertaistamisen vuoksi käytössä vain yksi testi ja siitä saadaan aikaiseksi seuraavanlainen oraakkelirajoite:

$\text{Or} \stackrel{\text{def}}{=} (x_1 = 0) \wedge (y_1 = 0) \wedge (\text{result} = 3)$.

Yhdistelmä $\varphi_{buggy} \wedge \text{Or}$ on toteutumaton, joka kuvastaa, että testi epäonnistuu. Useamman testin, esimerkiksi kahden testin, tapauksessa käytettäisiin uudelleen nimeämistä yhdistämisen kanssa ja yhdistelmästä tulisi tällöin $\texttt{Rename}(\varphi_{buggy} \wedge \text{Or}_1) \wedge \texttt{Rename}(\varphi_{buggy} \wedge \text{Or}_2)$ [Mechtaev et al., 2015].

Tavoitteena on löytää φ_{buggy} ilmaisut, joita täytyy muokata ja miten niitä pitää muokata. Muutokset ovat sellaisia, että muokattu kaava φ_{repair} tekee kaavasta $\varphi_{repair} \wedge Or$ toteutuvan. Tällöin kaava on

$\varphi_{repair} \stackrel{def}{=}$ (if $(x_1 \geq y_1)$ then $(y_2 = y_1 + 1)$ else $(y_2 = y_1 - 1)) \wedge$ (result = $y_2 + 2$), jossa siis ensimmäinen virheellinen ehto $x_1 > y_1$ on korjattu muotoon $x_1 \geq y_1$ [Mechtaev et al., 2015].

1	int foo(int x, int y) {
2	i f (x > y) / / FAULT: the conditional should be x >= y
3	y = y + 1;
4	else
5	y = y - 1;
6	return y + 2;
7	}
8	
9	void test_foo ( ) { assert (foo(0,0)==3); }

Taulukko 23. Esimerkki virheellisestä ohjelmasta ja sen testausfunktiosta, josta saadaan rajoite-oraakkeli.

DirectFix katkaisee korjattavasta kohdasta komponenttien välisiä yhteenliittymiä ja lisää korvaavia komponentteja ja niiden välisiä yhteenliittymiä. Yksinkertaisimman korjauksen etsimisen vuoksi pyritään katkaisemaan niin vähän yhteenliittymiä kuin mahdollista. Tämä saavutetaan pelkistämällä ohjelman korjausongelma osittaiseksi pMaxSMT-ongelman (*partial* MaxSMT) tapaukseksi eli instanssiksi [Mechtaev et al., 2015]. Alkuperäisessä lähteessä yhteenliittymiä käsiteltiin kuin piirikaaviona, josta pistettiin yhteyksiä poikki (*repair method views program as circuit*).

Korjauksen generoimiseksi, jotka perustuvat pMaxSMT:en, käytetään korjausehdoksi (*repair condition*) kutsuttua kaavaa. Jäljityskaavalle φ_{buggy} korjausehto φ_{rc} on seuraavanlainen

$\varphi_{rc} \stackrel{def}{=}$ (if v_1 then $(y_2 = v_2)$ else $(y_2 = v_3)) \wedge$ (result = v_4)

$\wedge$ cmpnt $(v_1 = x_1 > y_1) \wedge$ cmpnt $(v_2 = y_1 + 1)$

$\wedge$ cmpnt $(v_3 = y_1 - 1) \wedge$ cmpnt $(v_4 = y_2 + 2)$.

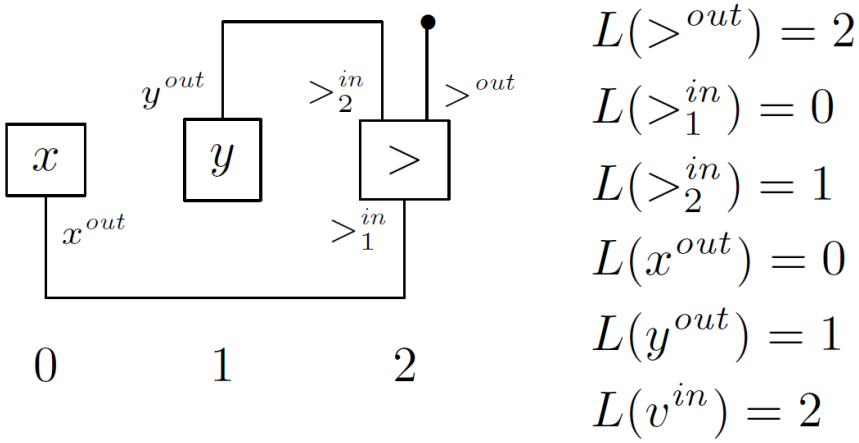
Kaava φ_{rc} on semanttisesti identtinen φ_{buggy} kanssa. Erona on, että φ_{buggy} oikeanpuolen ilmaisuja korvataan uusilla muuttujilla v_i, samalla pitäen voimassa yhtäsuuruus suhteen jokaisen v_i:n ja sen edustaman ilmaisun välillä cmpnt-funktion sisällä (esimerkiksi $v_1 = x_1 > y_1$). Funktio cmpnt komponentisoi parametri-ilmaisut yhteenliittymämuotoon seuraten komponenttipohjaisen syntetisoinnin ideaa [Mechtaev et al., 2015].

Korjauksen saamiseksi käytetään pMaxSMT-ratkaisijaa. Kaava jaetaan pMaxSMT:ssä heikkoihin ja tiukkoihin klausuuleihin. Tiukkoihin, eli aina toteutuviin klausuuleihin, sisällytetään komponenttien semantiikat ja oraakkelin data. Heikoilla klausuuleilla rajoitetaan ohjelman

ilmaisujen rakennetta, joista saadaan rakennerajoite (*structure constraint*). Esimerkiksi rakennerajoite ilmaisulle x>y, joka vastaa kuvan 4 kaaviota saadaan seuraavasti

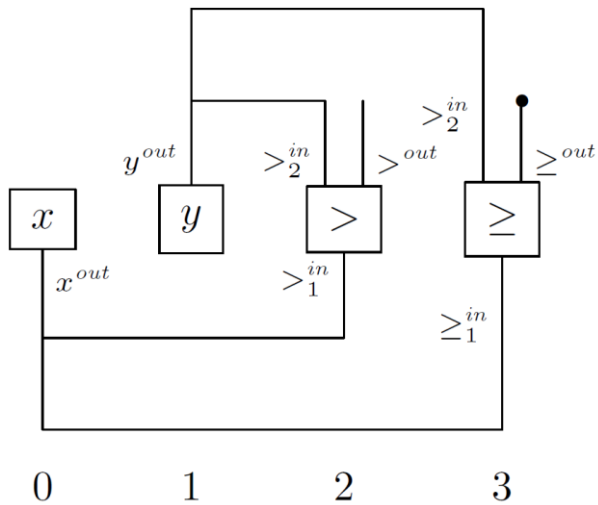
$$L(>_1^{in}) = L(x^{out}) \wedge L(>_2^{in}) = L(y^{out}) \wedge L(>^{out}) = L(v^{in}).$$

Rajoitteessa sidotaan ilmaisun tuloste uuteen muuttujaan v, eli v=x>y. Tämä rajoite spesifioi ilmaisun komponenttien väliset liittymät, kuten myös ilmaisun tulosteen sidoksen (*binding*). Kun kaava φ_{rc}∧Or on jaettu tiukkoihin ja heikkoihin klausuuleihin se syötetään pMaxSMT ratkaisijalle. Huomaa, että φ_{rc} ei ole sama kuin φ_{repair}. Ratkaisija poistaa tarvittaessa rakennerajoitteita ja palauttaa mallin, joka vastaa korjausta [Mechtaev et al., 2015].



Kuva 4. Ilmaisun x>y komponentit ja niiden väliset yhteydet. Oikealla paikkamuuttujien *L* joukko ja niiden arvoja. Komponentteja ja niiden välisiä yhteyksiä kuvaavan kuvan alla olevat numerot 0,1 ja 2 ovat mahdollisia paikkamuuttujien saamia arvoja [Mechtaev et al., 2015].

Ratkaisija pMaxSMT voi muokata ilmaisuja käyttämällä lisäkomponentteja. Kun kuvaa 5 verrataan kuvaan 4, huomataan, että yhteys komponentin > tulosteen ja v:n syötteen väliltä on katkaistu. Katkaistu yhteys vastaa rakenne rajoitetta $L(>^{out}) = L(v^{in})$. Uusia yhteyksiä on lisätty kolme kappaletta. Yksi x:n tulosteen ja komponentin ≥ syötteen väliin, y:n tulosteen ja ≥ toisen syötteen välille sekä ≥ tulosteen ja sitovan muuttujan v välille [Mechtaev et al., 2015].



Kuva 5. Ilmauksen x>y korjaus vaihtamalla komponentti > komponentilla ≥.

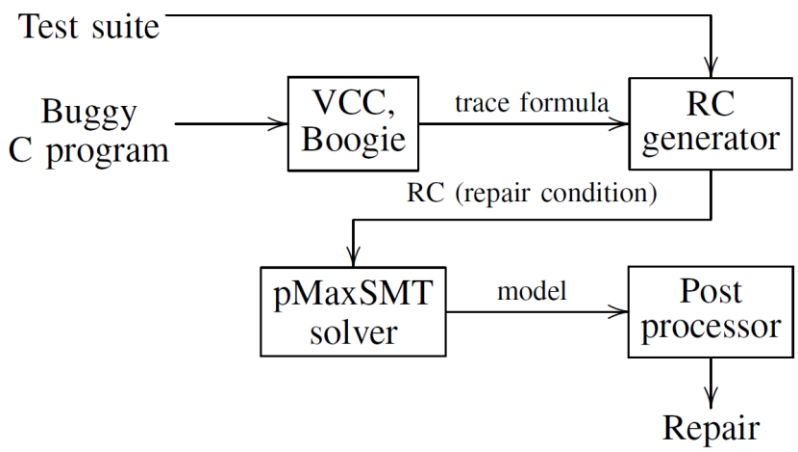
Etsimällä mallia, joka maksimoi toteutuneiden klausuulien $\varphi_{rc} \wedge Or$ määrän, tapahtuu liittymien katkaisu ja lisäys samanaikaisesti. Täten suoritetaan virheen paikantaminen ja korjauksen luonti samalla kertaa [Mechtaev et al., 2015].

Ohjelma	Yhteensä	DirectFix				SemFix			
		E	S	D	R	E	S	D	R
Tcas	30	16	29	2.26	12	3	11	4.1	17
Replace	5	5	5	2.8	0	3	4	10.2	2
Schedule	4	2	4	2.5	1	1	4	8.5	3
Schedule2	2	1	2	2	1	1	2	5	2
Coreutils	4	0	3	2	-	0	0	4	-
Overall	45(44)	53%	95%	2.31	31%	17%	46%	6.36	54%

Taulukko 24. Vertailua DirectFix ja SemFix korjaustyökalujen välillä. Sarake R ilmaisee korjausten aiheuttamien regressioiden määrän. Sarake E (Equal) ilmaisee, kuinka monta virheiden korjauksista vastaa ihmisten tekemiä korjauksia. Sarake S (Same location) ilmaisee, kuinka monessa korjauksessa korjattiin samaa kohtaa kuin ihmisten tekemissä korjauksissa. Sarake D (Differ) kuvaa korjausten aiheuttamien muutoksen määrää alkuperäiseen nähden AST puissa. Yhteensä sarakkeen overall kohdassa suluissa on alkuperäisen artikkelin virheellinen luku. Prosentit muihin sarakkeisiin tulevat oikein numerolla 45.

Heikkojen rajoitteiden käyttö vähentää syntetisointiaikaa. Tätä on perusteltu sillä, että SMT-ratkaisijan päälle kehitetty pMaxSMT-ratkaisija pystyy löytämään ratkaisun heikkoja rajoitteita sisältäviin kaavoihin. Sen sijaan SMT-ratkaisija ei pystynyt ratkaisemaan samoja kaavoja ilman heikkoja rajoitteita annetussa aikarajassa [Mechtaev et al., 2015].

Kun vertaillaan DirectFix:ä SemFix:n taulukossa 24, huomataan, että DirectFix korjaa useampia virheitä samalla tavalla kuin ihmiset. Merkittävimmillään ero on Tcas-ohjelmalla. DirectFix ei kuitenkaan ole yhtä nopea kuin SemFix ja Tcas-ohjelman korjaus vei keskimäärin DirectFixillä 3 minuuttia ja 20 sekuntia ja SemFixllä 9 sekuntia [Mechtaev et al., 2015].



Kuva 6. DirectFix:in komponentit ja suoritusjärjestys.

Työnkulku DirectFixin ja sen käyttämien kolmansien osapuolien sovellusten kanssa näkyy kuvassa 6. Näistä työkaluista VCC muuntaa C-kielisen ohjelman Boogie-ohjelmaksi. Boogie-tarkastaja (*Boogie verifier*) vastaanottaa syötteenä Boogie-ohjelman ja luo tarkastustilan φ_{vc} (*verification condition*), eli kaavan, jota käytetään osoittamaan, että tiettyä virhettä ei esiinny. Molemmat VCC ja Boogie pystyvät käsittelemään osoitinaritmetiikkaa. Boogien luoma tarkastustila on lähellä haluttua jäljityskaavaa. Tarkastustila taulukon 23 funktiolle foo on

$\neg((\text{if } (x_1 \geq y_1) \text{ then } (y_2 = y_1 + 1) \text{ else } (y_2 = y_1 - 1)) \Rightarrow (\text{result} = y_2 + 2))$.

Kaavasta saadaan jäljityskaava φ_{buggy}, kun otetaan φ_{vc}:stä negaatio ja korvataan $\Rightarrow$ konnektiivilla $\wedge$. Näiden pienien erojen takia $\varphi_{\text{buggy}} \wedge \psi_{test}$ on toteutumaton kun tarvitaan, ja taas $\varphi_{vc} \wedge \psi_{test}$ on toteutuva. DirectFixin kehittäjät muokkasivat Boogie-työkalua, jotta he saisivat jäljityskaavan tarkastustilan sijaan. Saatu jäljityskaava on SMT-LIB2-formaatissa selitettynä sijaintina lähdekoodissa [Mechtaev et al., 2015].

Viallisen ohjelman jäljityskaava ja sen testisarja syötetään DirectFixin korjausehto-generaattorille RC. Myöhemmin generoitu korjausehto syötetään pMaxSMT-ratkaisijalle. Malli, eli pMaxSMT:n antama ratkaisu, jälkikäsitellään korjauksen, eli paikkauksen, rakentamista varten [Mechtaev et al., 2015].

6. Virheiden paikantamiseen kehitetyt menetelmät

Tässä luvussa kerrotaan automaattisessa virheiden korjauksessa käytetyissä virheen paikantamisessa käytetyistä menetelmistä.

Painotettu polku on käytössä GenProgissa. Siinä korjattavana oleva ohjelma käännetään uudestaan siten, että jokaiseen AST-solmuun lisätään tieto siitä, missä solmussa ollaan. Nämä merkinnät kirjataan lokiin, kun onnistuneet ja epäonnistuneet testitapaukset ajetaan. Solmut, jotka ajetaan epäonnistuneessa testitapauksessa, mutta eivät onnistuneessa testitapauksessa, saavat korkeamman painoarvon [Weimer et al., 2010].

Yksi virheen todennäköisyyden arviointimittari on Tarantula error ranking. Se olettaa, että ne ohjelman kokonaisuudet, joita suoritetaan epäonnistuneissa testitapauksissa, ovat todennäköisemmin virheellisiä verrattuna alueisiin, joita suoritetaan onnistuneissa testitapauksissa. Tarantula kuitenkin huomioi mahdollisen virheellisen kohdan suorittamisen onnistuneissa testitapauksissa [Jones and Harrold, 2005].

Tarantulan toiminta selviää parhaiten Jonesin ja Haroldin antamalla esimerkillä, joka on esitelty taulukossa 25. Tarantula käyttää aluksi epäiltävyyskaavaa suoritettujen testitapausten tulosten perusteella:

$$Epäiltävyys = \frac{\frac{Rivin\ onnistuneet\ testit}{Kaikki\ onnistueet\ testit}}{\frac{Rivin\ onnistuneet\ testit}{Kaikki\ onnistueet\ testit} + \frac{Rivin\ epäonnistuneet\ testit}{Kaikki\ epäonnistueet\ testit}}.$$

Käyttämällä kaavaa esimerkiksi taulukon 25 riviin neljä saadaan seuraavat arvot. Rivin onnistuneet testit=3. Kaikki onnistuneet testit saadaan taulukossa 25 viimeisestä rivistä laskemalla onnistuneet testit yhteen, jolloin tuloksena on viisi. Rivin epäonnistuneet testit on 1, koska yksi epäonnistunut testitapaus suoritetaan rivillä. Esimerkissä ei ole enempää epäonnistuneita testitapauksia, joten kaikki epäonnistuneet testit on myös yksi. Sijoittamalla arvot kaavaan saadaan epäiltyvyydeksi 0.625, pyöristettynä 0.63. Lasketun epäiltävyyden ja epäilyksen alaisten koodilausumien avulla lasketaan virhe-arvo. Virhe-arvo on niiden rivien maksimimäärä, joilla on sama tai suurempi epäiltävyys kuin arvioitavalla rivillä. Taulukon 25 rivillä 2 virhe-arvo 7 saadaan seuraavasti. Kolmella rivillä on suurempi epäiltävyys kuin rivillä 2. Neljällä rivillä on sama epäiltävyys kuin rivillä 2. Siten virhe-arvoksi saadaan 3+4=7.

	Lausumat	Testitapaukset						Epäiltävyys	
Rivi	mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		Arvo
1	read("Enter 3 numbers:",x,y,z);	S	S	S	S	S	S	0.5	7
2	m = z;	S	S	S	S	S	S	0.5	7
3	if (y<z)	S	S	S	S	S	S	0.5	7
4	if (x<y)	S	S			S	S	0.63	3
5	m = y;		S					0.0	13
6	else if (x<z)	S				S	S	0.71	2
7	m = y; // *** bug ***	S					S	0.83	1
8	else			S	S			0.0	13
9	if (x>y)			S	S			0.0	13
10	m = y;			S				0.0	13
11	else if (x>z)				S			0.0	13
12	m = x;							0.0	13
13	print("Middle number is:",m);	S	S	S	S	S	S	0.5	7
}	Oikein/Väärin tulos testille.	O	O	O	O	O	V		

Taulukko 25. Esimerkki Tarantula-virheenpaikannuksen toiminnasta. Huomio testiaineiston järjestys suoritettaviin lauseisiin, joka on merkitty kirjaimella S.

Virheenpaikannuksen vaikutusta automaattiseen virheenkorjaukseen on selvitetty Durieux:n ja muiden [2015] raportissa. Siinä päädyttiin johtopäätökseen, että käytetty virheenpaikannusmenetelmä ei vaikuta merkittävästi automaattiseen virheiden korjaamiseen. Raportissa esitetään seitsemän eri virheenpaikannusmenetelmän algoritmit, ja mukana ovat Tarantula ja GenProgin käyttämä menetelmä.

7. Pohdintaa automaattisesta virheiden korjauksesta ja tutkielmasta

Lähtökohtani tämän tutkielman tekemiselle oli uteliaisuus automaattisesti tapahtuvaan virheiden korjaukseen, joka johtui siitä, että olin 4 vuotta työskennellyt ohjelmiston ylläpidossa. Mitä automaattinen virheiden korjaus on, miten se on toteutettu ja mitä sillä voidaan saavuttaa? Näihin en pysty vastaamaan tässä lyhyesti, vaan ohjaan lukijaa lukemaan tämän tutkielman ja tarvittaessa vielä siinä käytettävät lähteet. Menetelmät voivat saavuttaa noin 50 prosentin korjauskyvyn automaattisilla testeillä löydettyihin aina toistettaviin virheisiin. Kuinka suuri osa todellisista ohjelmistojen virheistä kuuluu tähän virheiden luokaan jää jatkotutkimukseksi. Lisäksi virheiden korjaajat saattavat tarvita kolmansien osapuolten työkaluja, joissa on omat rajoitteensa, jotka rajaavat mahdollisia korjattavia virheitä. Missään tapauksessa manuaalista virheiden korjausta ei voida täysin korvata.

Onko jokin menetelmä selvästi muita tehokkaampi? Vastaus perustuu vain lähteissä mainittujen tulosten ja menetelmien testauksessa käytettyjen testiaineistojen silmämääräiseen analysointiin. Menetelmillä saatujen tulosten vertailu keskenään ei anna oikeaa kuvaa niiden paremmuudesta. Kuitenkin on nähtävissä, että mikään menetelmä ei ole ylivertainen muihin. Jokaisessa menetelmässä on tyydyttävä jonkinlaiseen kompromissiin. Semanttiset menetelmät löytävät yksinkertaisiin korjauksiin laadukkaan korjauksen, koodirivejä lisäävissä tapauksissa, geneettiseen algoritmiin verrattuna. Geneettinen algoritmi taas voi toimia myös monimutkaisempien virheiden korjauksessa. DirectFix tekee parempia korjauksia kuin SemFix, mutta kuluttaa korjaamiseen kertaluokkaa enemmän aikaa. Tämä ei kuitenkaan ole merkittävä tekijä, kun otetaan huomioon, että automaattinen virheiden korjaus tultaisiin kuitenkin tekemään niin sanoittuina hiljaisina tunteina. DirectFix:n voi kuitenkin ajatella olevan Semfix:n uudempi versio. Samoin GenProgin uudempi versio toimii vanhaa paremmin.

Onko vaikutuksia testauksen suunnitteluun tai ohjelmistotuotannon koulutukseen? Automaattisen virheidenkorjauksen käyttöönotto ei vaadi muuta testausautomaatiota enempää koulutusta. Käyttöönotto voi ohjata testitapausten suunnittelua. Tähän liittyy osittain myös jatkokysymys, kannattaako ohjelmistoprojekteissa ottaa automaattinen virheidenkorjaus tuotantokäyttöön? Automaattinen virheidenkorjaus todennäköisesti tarjoaisi jonkinasteista hyötyä hieman isompien projektien tuotantokäytössä. Menetelmiä voisi käyttää rinnan hiljaisten tuntien automaattisen regressiotestauksen kanssa. Tässä mielessä näkisin, että joissakin projekteissa automaattista virheidenkorjausta aiotaan käyttää testausmielessä lähiaikoina. Hyödyt eivät kuitenkaan tule olemaan merkittäviä, koska vaikeammat, esimerkiksi harvoin ja epädeterministisesti toistuvat virheet, jäävät edelleen käsintehtäväksi työksi. Sen sijaan aloittavien ohjelmoijien koulutukseen automaattiset virheidenkorjaus menetelmät toimivat hyvin, mikäli virheellisen koodin tehnyt ohjelmoija saa nähdä korjaustyökalun tekemän korjauksen. Lisäksi mitään automaattista virheidenkorjaus työkalua, jonka saa helposti integroitua mukaan tuotantojärjestelmäään, ei tällä hetkellä ole saatavilla, vaan tarjolla on kokeellisia ohjelmia.

7.1. Tutkielman tekemisessä kohdattuja ongelmia

Tutkimusta tehdessä oli vaikeuksia selvittää aihepiirin historiaa. Ennen 2000-lukua kiinnostus on ollut ennemmin automaattisessa ohjelman luonnissa, ja tutkimuksia virheiden automaattisesta korjauksesta ei ole tehty. Samoin termien selvitystyö vei oletettua enemmän aikaa. Termien osalta ongelmina olivat, että niitä ei ole suomennettu, ja osa termistöstä mainitaan lähteissä vain lyhenteinä, mutta samalle lyhenteelle on useampi itse tietojenkäsittelytieteisiin liittyvä merkitys. Lisäksi tietyt matemaattiset englanninkieliset lauseet ja fraasit kääntyvät huonosti suomenkielisiksi matemaattisiksi lauseiksi. Tätä vaikeutta hankaloitti vielä oma kokemattomuus ja taustatietojen puute matematiikan alueelta.

7.2. Jatkotutkimuksia ja käsittelemättömiä asioita

Formaalien menetelmien ja tietokoneiden nopeuden kehittyminen antaisi perusteluja käyttää formaaleja menetelmiä ohjelmistotuotannossa paljon nykyistä enemmän. Niiden käyttäminen vaatii kuitenkin huomattavasti erilaisempaa koulutusta verrattuna ohjelmistojen toteutukseen ja testaukseen jotakin ohjelmointikieltä käyttäen. Lisäksi suuret ohjelmistosuunnittelutyökaluja valmistavat kaupalliset yritykset eivät panosta formaalien menetelmien käyttämisen houkuttelevuuteen, eivätkä täten ole kehittämässä vastaavan tasoisia työkaluja mitä on saatavilla esimerkiksi imperatiivisille ohjelmointikielille. Täten formaalit menetelmät ovat vieläkin vähemmän kiinnostavia vaihtoehtoja. Kun tähän lisätään vielä se seikka, että formaalit menetelmät eivät takaa täyttä virheettömyyttä, jäävät kyseiset menetelmät vielä vain tutkimuksen ja harvojen kaupallisten projektien käyttöön. Tämä on vain subjektiivinen näkemykseni lukemieni artikkelien perusteella tätä tutkielmaa tehdessäni. Formaalien menetelmien osalta voisi tutkia erilailla tapahtuvan koulutuksen mahdollisuuksia ja asian selkeämpää esitystapaa.

Tässä tutkimuksessa jäi käsittelemättä semanttiseen analyysiin perustuva Nopol, joka on kehitetty korjaamaan ehtolauseiden viallisia ehtoja [DeMarco et al., 2014]. Nopolin kehittäjät kiinnittivät myös huomiota testiaineiston valinnan vaikutukseen verrattaessa eri menetelmien tehokkuutta. Geneettiseen ohjelmointiin perustuva MUT-APR [Assiri and Bieman, 2014] olisi ollut erilainen lähestymistapa geneettiseen ohjelmointiin verrattuna GenProgiin. MUT-APR ei käytä korjattavan ohjelman koodia, vaan sisältää joukon sisäisiä operaatioita, joita se sitten lisää mutaatioilla korjattavaan koodiin. Eiffel-kielen sopimuksiin perustuva menetelmä AutoFix [Pei et al., 2014] luo korjauksen käyttäen hyväkseen funktion esi- ja jälkiehtoja. AutoFix on siis tavallaan spesifikaatioista korjauksen kehittävä menetelmä. DirectFix:n [Mechtaev et al., 2015] menetelmän käsittelin yksinkertaisesti, ja en esitellyt menetelmän monimutkaisia yksityiskohtia, jotka liittyvät ohjelmasilmukoihin ja useamman ohjelmarivin korjauksiin.

Jatkotutkimuksena voisi olla eri menetelmien henkilökohtainen testaus. GenProgin ja Nopolin lähdekoodi on saatavissa, ja itse tehdyillä testeillä voisi mahdollisesti löytää uusia ongelmakohtia automaattiseen virheiden korjaamiseen. Testiaineiston vaikutuksesta ja korjausten kelvollisuudesta

voi tehdä kuitenkin arvioita Durieux: ja muiden [2015] raportista. Eri menetelmiä ei myös ole vielä koitettu yhdistää, mutta mahdollisuudesta on mainittu muutamissa lähteissä.

Viiteluettelo

[Al-Ekram et al., 2005] Raihan Al-Ekram, Archana Adma and Olga Baysal, diffX: an algorithm to detect changes in multi-version XML documents, In: *Proc. of the 2005 Conference of the Centre for Advanced Studies on Collaborative research* (CASCON), IBM Press, (2005), 1-11.

[Assiri and Bieman, 2014] Fatmah Yousef Assiri and James M. Bieman, An assessment of the quality of automated program operator repair, In: *Proc. of the 7th International Conference on Software Testing, Verification and Validation* (ICST), IEEE, (2014), 273-282.

[Balzer, 1985] Robert Balzer, A 15 year perspective on automatic programming, *IEEE Transactions on Software Engineering*, **11**, 11 (Nov 1985), 1257-1268.

[Baxter et al., 1998] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna and Lorraine Bier, Clone detection using abstract syntax trees, In: *Proc. of the International Conference of Software Maintenance*, IEEE, (Nov 1998), 368-377.

[Cok, 2013] David R Cok, *The SMT-LIBv2 Language and Tools: A Tutorial*, GrammaTech Inc., Version 1.2.1, November 23, 2013.

[Cytron et al., 1989] Ron Cytron, Jeanne Ferrante, Barry, K. Rosen, Mark N. Wegman and F. Kenneth Zadeck, An efficient method of computing static single assignment form, In: *Proc. of the 16th Symposium on Principles of Programming Languages* (POPL). ACM, (1989), 25-35.

[DeMarco et al., 2014] Favio DeMarco, Jifeng Xuan, Daniel Le Berre and Martin Monperrus, Automatic repair of buggy if conditions and missing preconditions with SMT, In: *Proc. of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis* (CSTVA). ACM, (May 2014), 30-39.

[Durieux et al., 2015] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan, Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset, (May 2015).

[Forrest et al., 2013] Stephanie Forrest, Claire Le Goues and Westley Weimer, Current challenges in automatic software repair, *Journal of Software Quality*, **21**, 3 (Sep 2013), 421-443.

[Goues et al., 2012] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest and Westley Weimer, GenProg: A generic method for automatic software repair, *IEEE Transactions on Software Engineering*, **38**, 1 (Jan/Feb 2012), 54-72.

[Gulwani et al., 2010] Sumit Gulwani, Susmit Jha, Ashish Tiwari and Ramarathnam Venkatesan, Component based synthesis applied to bitvector circuits, Technical Report MSR-TR-2010-12, Microsoft Research, (Feb 2010).

[Gurari, 1989] Eitan M. Gurari, *An Introduction to the Theory of Computation*, Computer Science Press, 1989.

[Jha et al., 2010] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia and Ashish Tiwari, Oracle-guided component-based program synthesis, In: *Proc. of the 32nd International Conference on Software Engineering* (ICSE), IEEE/ACM, (2010), 215-224.

[Jones and Harrold, 2005] James A. Jones and Mary Jean Harrold, Empirical evaluation of the tarantula automatic fault localization technique, In: *Proc. of the 20th International Conference on Automated Software Engineering* (ASE), IEEE/ACM, (2005), 273-282.

[Järvisalo, 2016] Matti Järvisalo, Satisfiability, Boolean Modeling and Computation, University of Helsinki, January 19, 2016.

[Järvisalo, 2004] Matti Järvisalo, Lauselogiikan toteutuvuustarkastus: käytännönläheistä teoriaa, Tietojenkäsittelytiede, **22**, Joulukuu, 2004, 47-63.

[Kaarakka ja Lätti, 2006] Terhi Kaarakka ja Isto Lätti, Algoritmimatematiikka, Tampereen teknillinen yliopisto, syksy 2006.

[Kaijanaho, 2006] Antti-Juhani Kaijanaho, Käytännön formaalit menetelmät, Luentokalvot Johdatus ohjelmistotekniikkaan -kurssilta, Jyväskylän yliopisto, Marraskuu 21, 2006. Saatavissa: http://mit.jyu.fi/opetus/kurssit/jot/2006/luennot/jot-fm-2006.pdf. [Haettu 19.11.2016].

[Kung and Zhu, 2008] David Kung and Hong Zhu, Software verification and validation, Wiley Encyclopedia of Computer Science and Engineering, John Wiley & Sons, Inc., 2008.

[Kähkönen, 2009] Kari Kähkönen, Automated test generation for software components, TKK Reports in Information and Computer Science 26, Helsinki University of Technology, 2009.

[Leavens and Cheon, 2006] Gary T. Leavens and Yoonsik Cheon, Design by contract with JML, http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf, (2006).

[Lynce and Ouaknine, 2006] Inês Lynce and Joël Ouaknine, Sudoku as a SAT Problem, In: *Proc. of the 9th International Symposium on AI and Mathematics*, (January 2006).

[Moura et al., 2007] Leonardo De Moura, Bruno Dutertre and Natarajan Shankar, A tutorial on satisfiability modulo theories, In: *Proc. of the 19th International Conference on Computer Aided Verification* (CAV), Springer-Verlag Berlin, (July 2007), 20-36.

[Necula et al., 2002] George C. Necula, Scott McPeak, Shree Prakash Rahul and Westley Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs, In: *Proc. of the 11th International Conference on Compiler Construction* (CC), Springer-Verlag London, (2002), 213-228.

[Nguyen et al., 2013] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury and Satish Chandra, SemFix: Program repair via semantic analysis, In: *Proc. of the 35th International Conference on Software Engineering* (ICSE), IEEE, (2013), 772-781.

[Pei et al., 2014] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer and Andreas Zeller, Automated fixing of programs with contracts, *IEEE Transactions on Software Engineering*, **40**, 5 (May 2014), 427-449.

[Rauhala, 2010] Eija Rauhala, Ohjelmistotestauksen suunnittelu – Case: A –lehdet Oy:n laskujen tulostusohjelma, Opinnäytetyö, Haaga-Helia ammattikorkeakoulu, 2010.

[Salakoski, 2002] Harri Salakoski, Neuroverkkojen tuottaminen geneettisen algoritmin avulla, Pro gradu -tutkielma, Tampereen yliopisto, 2002.

[Torlak et al., 2011] Emina Torlak, Satish Chandra, Shaon Barman and Rastislav Bodik, Angelic debugging, In: *Proc. of the 33rd International Conference on Software Engineering* (ICSE), IEEE/ACM, (2011), 121-130.

[Vehkaperä, 2010] Kaisa-Mari Vehkaperä, Testauksen automatisointiprosessin kehittäminen terveydenhuollon tietojärjestelmään, Opinnäytetyö, Oulun seudun ammattikorkeakoulu, 20101.

[Välimäki, 2011] Tuomas Välimäki, Automaattisen testausjärjestelmän kehitys, Diplomityö, Tampereen teknillinen yliopisto, 2011.

[Weimer et al., 2010] Westley Weimer, Stephanie Forrest, Claire Le Goues and ThanhVu Nguyen, Automatic program repair with evolutionary computation, *Comm. ACM*, **53**, 5 (May 2010), 109-116.

[Mechtaev et al., 2015] Sergey Mechtaev, Nguyen, Jooyong Yi and Abhik Roychoudhury, DirectFix: Looking for simple program repairs, In: *Proc. of the 37th International Conference on Software Engineering* (ICSE), IEEE, (2015), 448-458.

[Zeller and Hildebrandt, 2002] Andreas Zeller and Ralf Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering*, **28**, 2 (February 2002), 183-200.